

A First Course in COMPUTATIONAL PHYSICS



Paul L. DeVries

A First Course in Computational Physics

Paul L. DeVries

Miami University

Oxford, Ohio



JOHN WILEY & SONS, INC.

NEW YORK · CHICHESTER · BRISBANE · TORONTO · SINGAPORE

ACQUISITIONS EDITOR: *Cliff Mills*
MARKETING MANAGER: *Catherine Faduska*
PRODUCTION EDITOR: *Sandra Russell*
MANUFACTURING MANAGER: *Inez Pettis*

This book was set in Century Schoolbook by the author and printed and bound by Hamilton Printing Company. The cover was printed by New England Book Components, Inc.

Recognizing the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc. to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

Copyright ©1994 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress cataloging in publication data:

DeVries, Paul L., 1948 –

A first course in computational physics / Paul L. DeVries.

p. cm.

Includes index.

ISBN 0-471-54869-3

1. Physics — Data processing. 2. FORTRAN (Computer program language). 3. Mathematical Physics. I. Title

QC52.D48 1993

530'.0285'5133 — dc20

93-15694

CIP

Printed in the United States of America

10 9 8 7 6 5 4 3 2

To Judi, Brandon, and Janna.

Preface

Computers have changed the way physics is done, but those changes are only slowly making their way into the typical physics curriculum. This textbook is designed to help speed that transition.

Computational physics is now widely accepted as a third, equally valid complement to the traditional experimental and theoretical approaches to physics. It clearly relies upon areas that lie some distance from the traditional physics curriculum, however. In this text, I attempt to provide a reasonably thorough, numerically sound foundation for its development. However, I have not attempted to be rigorous; this is not meant to be a text on numerical analysis. Likewise, this is not a programming manual: I assume that the student is already familiar with the elements of the computer language, and is ready to apply it to the task of scientific computing.

The FORTRAN language is used throughout this text. It is widely available, continually updated, and remains the most commonly used programming language in science. Recent FORTRAN compilers written by Microsoft provide access to many graphics routines, enabling students to generate simple figures from within their FORTRAN programs running on PC-compatible microcomputers. Such graphics capabilities greatly enhance the value of the computer experience.

The various chapters of the text discuss different types of computational problems, with exercises developed around problems of physical interest. Topics such as root finding, Newton–Cotes integration, and ordinary differential equations are included and presented in the context of physics problems. These are supplemented by discussions of topics such as orthogonal polynomials and Monte Carlo integration, and a chapter on partial differential equations. A few topics rarely seen at this level, such as computerized tomography, are also included. Within each chapter, the student is led from relatively elementary problems and simple numerical approaches through derivations of more complex and sophisticated methods, often culminating in the solution to problems of significant difficulty. The goal is to demonstrate how numerical methods are used to solve the problems that physicists face. The text introduces the student to the *process* of approaching problems from a computational point of view: understanding the physics and describing it in mathematical terms, manipulating the mathematics to the point where a nu-

merical method can be applied, obtaining a numerical solution, and understanding the physical problem in terms of the numerical solution that's been generated.

The material is intended for the student who has successfully completed a typical year-long course in university physics. Many of the topics covered would normally be presented at a later stage, but the computational approach enables the student to apply more than his or her own analytic tools to the problem. It is unlikely, however, that any but the most serious of students will complete the text in one semester. There is an abundance of material, so that the instructor can choose topics that best meet the needs of the students.

A First Course in Computational Physics is the result of several years of teaching. Initially, handwritten notes to accompany my lectures were distributed, and as more material was compiled and old notes rewritten, an outline of the text developed. The text has been thoroughly tested, and, needless to say, I want to thank all my students for their involvement in the project. In particular, I am grateful to Jane Scipione, Chris Sweeney, Heather Frase, Don Crandall, Jeff Kleinfeld, and Augusto Catalan for their numerous contributions. I also want to thank several of my colleagues, including Larry Downes, Comer Duncan, Ian Gatland, Donald Kelly, Philip Macklin, and Donald Shirer, for reading and commenting on the manuscript. Any errors that remain are, of course, my own doing. The author invites any and all comments, corrections, additions, and suggestions for improving the text.

*Paul L. DeVries
Oxford, Ohio
January 1993*

Contents

Chapter 1: Introduction 1

FORTRAN — the Computer Language of Science	2
Getting Started	3
Running the Program	5
A Numerical Example	8
Code Fragments	11
A Brief Guide to Good Programming	13
Debugging and Testing	19
A Cautionary Note	20
Elementary Computer Graphics	21
And in Living Color!	26
Classic Curves	27
Monster Curves	29
The Mandelbrot Set	33
References	39

Chapter 2: Functions and Roots 41

Finding the Roots of a Function	41
Mr. Taylor's Series	51
The Newton–Raphson Method	54
Fools Rush In ...	58
Rates of Convergence	60
Exhaustive Searching	66
Look, Ma, No Derivatives!	67
Accelerating the Rate of Convergence	71
A Little Quantum Mechanics Problem	74
Computing Strategy	79
References	85

Chapter 3: Interpolation and Approximation 86

Lagrange Interpolation	86
The Airy Pattern	89
Hermite Interpolation	93

Cubic Splines	95
Tridiagonal Linear Systems	99
Cubic Spline Interpolation	103
Approximation of Derivatives	108
Richardson Extrapolation	113
Curve Fitting by Least Squares	117
Gaussian Elimination	119
General Least Squares Fitting	131
Least Squares and Orthogonal Polynomials	134
Nonlinear Least Squares	138
References	148

Chapter 4: Numerical Integration **149**

Anaxagoras of Clazomenae	149
Primitive Integration Formulas	150
Composite Formulas	152
Errors... and Corrections	153
Romberg Integration	155
Diffraction at a Knife's Edge	157
A Change of Variables	157
The "Simple" Pendulum	160
Improper Integrals	164
The Mathematical Magic of Gauss	169
Orthogonal Polynomials	171
Gaussian Integration	173
Composite Rules	180
Gauss–Laguerre Quadrature	180
Multidimensional Numerical Integration	183
Other Integration Domains	186
A Little Physics Problem	188
More on Orthogonal Polynomials	189
Monte Carlo Integration	191
Monte Carlo Simulations	200
References	206

Chapter 5: Ordinary Differential Equations **207**

Euler Methods	208
Constants of the Motion	212
Runge–Kutta Methods	215
Adaptive Step Sizes	218
Runge–Kutta–Fehlberg	219
Second Order Differential Equations	226

The Van der Pol Oscillator	230
Phase Space	231
The Finite Amplitude Pendulum	233
The Animated Pendulum	234
Another Little Quantum Mechanics Problem	236
Several Dependent Variables	241
Shoot the Moon	242
Finite Differences	245
SOR	250
Discretisation Error	250
A Vibrating String	254
Eigenvalues via Finite Differences	257
The Power Method	260
Eigenvectors	262
Finite Elements	265
An Eigenvalue Problem	271
References	278

Chapter 6: Fourier Analysis **280**

The Fourier Series	280
The Fourier Transform	284
Properties of the Fourier Transform	286
Convolution and Correlation	294
Ranging	304
The Discrete Fourier Transform	309
The Fast Fourier Transform	312
Life in the Fast Lane	316
Spectrum Analysis	319
The Duffing Oscillator	324
Computerized Tomography	325
References	338

Chapter 7: Partial Differential Equations **340**

Classes of Partial Differential Equations	340
The Vibrating String... Again!	342
Finite Difference Equations	344
The Steady-State Heat Equation	354
Isotherms	358
Irregular Physical Boundaries	359
Neumann Boundary Conditions	361
A Magnetic Problem	364
Boundary Conditions	366

The Finite Difference Equations	367
Another Comment on Strategy	370
Are We There Yet?	374
Spectral Methods	374
The Pseudo–Spectral Method	377
A Sample Problem	385
The Potential Step	388
The Well	391
The Barrier	394
And There’s More...	394
References	394

Appendix A: Software Installation **396**

Installing the Software	396
The FL Command	398
AUTOEXEC.BAT	400
README.DOC	401

Appendix B: Using FCCP.lib **402**

Library User’s Guide	403
----------------------	------------

Appendix C: Library Internals **407**

Library Technical Reference	407
-----------------------------	------------

Index **421**

Chapter 1:

Introduction

This is a book about physics — or at least, about how to do physics. The simple truth is that the computer now permeates our society and has changed the way we think about many things, including science in general and physics in particular. It used to be that there was theoretical physics, which dealt with developing and applying theories, often with an emphasis on mathematics and “rigor.” There was experimental physics, which was also concerned with theories, and testing their validity in the laboratory, but was primarily concerned with making observations and quantitative measurements. Now there is also computational physics, in which numerical experiments are performed in the computer laboratory — an interesting marriage of the traditional approaches to physics. However, just as the traditional theoretician needs a working knowledge of analytic mathematics, and the traditional experimentalist needs a working knowledge of electronics, vacuum pumps, and data acquisition, the computational physicist needs a working knowledge of numerical analysis and computer programming. Beyond mastering these basic tools, a physicist must know how to use them to achieve the ultimate goal: to understand the physical universe.

In this text, we’ll discuss the tools of the computational physicist, from integrating functions and solving differential equations to using the Fast Fourier Transform to follow the time evolution of a quantum mechanical wavepacket. Our goal is not to turn you into a computational physicist, but to make you aware of what is involved in computational physics. Even if you, personally, never do any serious computing, it’s necessary for you to have some idea of what is reasonably possible. Most of you, though, will find yourselves doing some computing, and you are likely to use some aspects of what’s presented here — canned programs rarely exist for novel, interesting physics, and so we have to write them ourselves! We hope to provide you with some useful tools, enough experience to foster intuition, and a bit of insight into how physics in your generation will be conducted.

In this chapter we’ll introduce you to programming and to a philosophy of programming that we’ve found to be useful. We’ll also discuss some of

the details of editing files, and compiling and running FORTRAN programs. And most importantly, we'll present a brief description of some of the elements of good programming. This discussion is not intended to transform you into a computer wizard — simply to help you write clear, reliable programs in a timely manner. And finally, we will use simple computer graphics to help solve physics problems. The capability of *visualizing* a numerical solution as it's being generated is a tremendous tool in understanding both the solution and the numerical methods. Although by no means sophisticated, our graphics tools enable the user to produce simple line plots on the computer screen easily.

FORTRAN — the Computer Language of Science

There's a large variety of computer languages out there that could be (and are!) used in computational physics: BASIC, C, FORTRAN, and PASCAL all have their ardent supporters. These languages have supporters because each has its particular merits, which should not be lightly dismissed. Indeed, many (if not most) computing professionals know and use more than one programming language, choosing the best language for the job at hand. Generally speaking, physicists are usually not "computing professionals." We're not interested in the mechanics of computing except when it directly relates to a problem we're trying to solve. With this in mind, there's a lot to be said for using only one language and learning it well rather than using several languages without being really fluent in any of them.

So that's the argument for a single language. If a need later develops and you find it necessary to learn a second language, then so be it. But why should the first language be FORTRAN? If you take a formal programming course, you will almost certainly be told that FORTRAN is not a good language to use. However, it's also the language in which the majority of scientific and engineering calculations are performed. After all, FORTRAN was developed for FORmula TRANslation, and it's very good at that. In the physical sciences, it's been estimated that 90% of all computer time is spent executing programs that are written in FORTRAN. So, why FORTRAN? Because it's *the* language of science!

While we don't agree with all the criticisms of FORTRAN, we don't ignore them either — we simply temper them with our own goals and expectations. As an example, we're told that *structure* in programming leads to *inherently more reliable* programs that are easier to read, understand, and modify. True enough. The value of top-down design, the virtue of strong typing, and the potential horrors of GOTO's have also been impressed upon us.

We attempt to follow the tenets of good programming style, for the simple reason that it does *indeed* help us produce better programs. And while we're aware that the language is lacking in some areas, such as data structures, we're also aware of unique advantages of the language, including the ability to pass function names in a parameter list. On balance, FORTRAN is not such a bad choice.

Neither is FORTRAN set in stone. In response to advances in hardware and developments in other languages, the American National Standards Institute (ANSI) established a committee, X3J3, to consider a new FORTRAN standard. This new standard, now known as FORTRAN 90, incorporates many of the innovations introduced in other languages, and extends its data handling and procedural capabilities. While it might be argued that other languages presently have advantages over FORTRAN, many of these features will be incorporated into the new standard and, hence, the perceived advantages will evaporate. Since the adoption of the new standard, there has been discussion of the development of yet another FORTRAN standard, one that would take advantage of the fundamental changes taking place in computer hardware with regard to massively parallel computers. In all likelihood, FORTRAN will be around for a long time to come.

This text assumes that you are FORTRAN literate — that you are aware of the basic statements and constructs of the language. It does not assume that you are fluent in the language or knowledgeable in scientific programming. As far as computer programming is concerned, this text is primarily concerned with how to perform scientific calculations, not how to code a particular statement.

Getting Started

Before we're through, you'll learn many things about computing, and a few things about computers. While most of what we have to say will be pertinent to any computer or computing system, our specific comments and examples in this text are directed toward microcomputers. Unfortunately, you need to know quite a bit to even begin. There are operating systems, prompts, commands, macros, batch files, and other technical details *ad nauseam*. We assume that you already know a little FORTRAN, are reasonably intelligent and motivated, and have access to an IBM-style personal computer equipped with a Microsoft FORTRAN compiler. We will then introduce you to the technicalities a little at a time, as they are needed. A consequence of this approach is that you may never learn all the details, which is just fine for most of you; we adopt the attitude that you should be required to know as few of these

technical details as possible, *but no fewer!* If you want to know more, the reference manuals contain all the dry facts, and your local computer wizard (and there is *always* a local wizard) can fill you in on the rest. It has been said that a journey of a thousand miles begins with a single step, and so it is that we must *start*, someplace. Traditionally, the first program that anyone writes is one that says “hello”:

```
write(*,*)'Hello, world'
end
```

Now, admittedly this program doesn't do much — but at least it doesn't take long to do it! There are many different ways of writing output, but the WRITE statement used here is probably the simplest. The first * informs the computer that it should write to the *standard input/output* device, which is just the screen. (In a READ statement, an * would inform the computer to read from the keyboard.) It is also possible to write to the printer, or to another file, by substituting a *unit number* for the *. The second * tells the computer to write in its own way, which is OK for right now. Later, we will specify a particular FORMAT in which the information is to be displayed. And what we want displayed is just the message contained between the two single quote marks. Numerical data can also be displayed using the *,* construction, by simply giving the name of the variable to be printed.

To execute this program, several steps must be accomplished, beginning with entering this FORTRAN *source code* into the computer. Actually, we enter it into a *file*, which we choose to call HELLO.FOR. For our own benefit, file names should be chosen to be as descriptive as possible, but they are limited to 8 characters. The 3-character extension after the period informs the computer of the nature of the file, and must be .FOR for a FORTRAN source file.

There are several ways to create the file; the most generally useful one is with a text editor of some kind. If you are already familiar with an editor that produces standard ASCII output, by all means use it. (Most word processors embed special characters in the output file, which would prevent the file from being used by the FORTRAN compiler. Fortunately, most of them can be configured to produce standard ASCII output.) If you don't have a favorite editor, I strongly suggest that you choose a full-screen editor such as the one Microsoft distributes with its FORTRAN compiler. Whichever editor you choose, you need to become familiar with its usage so that program entry is not an obstacle. Let's assume that your editor of choice is named E. (E is just a fictitious name for an editor. You'll need to substitute the real name of your particular editor in all that follows.) To invoke this editor, type:

```
E HELLO.FOR
```

This instructs the computer to run the program named E, which is your text editor, and it tells the program E that you want to work with a file named HELLO.FOR. Had the file already existed, E would provide you with a copy of it to modify. Since it doesn't yet exist, E will create it for you.

The editor creates the file, and then displays a mostly blank screen: the contents of your file. You can now insert text by typing it in, and you can move around the screen using the arrow keys. Once you have more than a full screen of text, you can move up or down a page at a time with the PgUp and PgDn keys, and Home will (usually) take you to the top of the file. Beyond these rudimentary commands, you need to learn the specific commands for your particular editor. *The very first command you should learn is the specific one for your editor that allows you to quit the editor and save your work.*

Since FORTRAN code must begin in column 7 or greater, you can use the space bar to position the cursor in the proper column. (With many editors, the tab key can also be used to move the cursor.) Microcomputers and modern program editors are pretty smart, so that the editor probably knows that computer code is often indented — FORTRAN requires the 6-space indentation, and you will also want to indent portions of the code to aid in its readability. We'll assume that E knows about indentation. So space over to column 7, type `write(*,*) 'Hello, world'`, and press the return key. Note that E moves you down a line and positions the cursor in column 7. You're ready to type the next line.

Many (if not most) modern compilers, including Microsoft FORTRAN, are insensitive to case. That is, the compiler makes no distinction between a lowercase `write` and an uppercase `WRITE`. That's a nice feature, since we can then use lowercase for most of the FORTRAN statements and reserve the uppercase to denote IMPORTANT or CRUCIAL statements, thus adding to the readability of the computer code. By the way, the operating system is also case insensitive, so that we could have typed `e hello.for` with exactly the same results. (Note: the "old" FORTRAN standard required all uppercase characters. The "new" standard allows uppercase and lowercase, but is insensitive to it.)

Running the Program

There are several steps that must be performed before the FORTRAN source code you've typed into HELLO.FOR is ready to execute. The first thing is to

convert your code into the ones and zeros that the computer understands; this is called *compiling* your program, and is done by (of all things) the compiler. The result of the compilation is an *object* file with the name `HELLO.OBJ`. Your code will undoubtedly need other information as well, and so it must be *linked* with certain system files that contain that information. For example, sines and cosines are provided as separate pieces of code that must be combined — linked — with yours if you require those functions. The output of the linker is an *executable* file named `HELLO.EXE`. For our example, this process is not particularly complicated. However, it will become more complicated as we look at the exercises in this text that need additional information specified to both the compiler and the linker. These steps must be performed every time a change is made in the FORTRAN source code and are virtually identical for all the programs you will want to run. To enter all the information manually, however, would be extremely tedious and susceptible to error. Fortunately, there is an easy way for us to specify all the necessary information, using environment variables. This is described in detail in Appendix A. After having followed the instructions outlined there, you can compile and link your FORTRAN programs by simply typing, for example,

```
f1 hello.for
```

This is the command to the operating system to compile (and link) your program. (A fuller description of this command is given in Appendix A.) A message will appear on the screen, identifying the specific version of the FORTRAN compiler being used, and the name of the file being compiled, e.g., `hello.for`. Finally, some information about the LINKER will be displayed. Note that it is necessary to include the `.for` extension. As written, the `f1` command compiles and links the program, producing an *executable* file. Should you accidentally omit the `.FOR` extension, only the linking step would be performed.

To run the program (that is, to *execute* it), you type the filename, `hello`, followed by the Enter key. (Note that you *do not* type `“for”` after the filename to execute the program.) Assuming there are no errors, the program will run and the “hello, world” message will be written to your screen. It’s done! (I hope. If the computer didn’t perform as I’ve indicated, make sure you that you’ve followed all the instructions precisely — computers can be quite picky about every detail. If the program still doesn’t run, see the local wizard.)

We are primarily concerned with the process of solving problems, using the computer. However, we need to say a little about the computer code itself. Over time “the code”, e.g., a collection of specific FORTRAN statements, tends to take on a life of its own. Certainly, it’s to no one’s advantage

to “reinvent the wheel.” A particular program (or even a portion of a program) that is of proven value tends to be used over and over again, often modified or “enhanced” to cover a slightly different circumstance. Often these enhancements are written by different people over an extended period of time. In order to control the growth of the code through orderly channels, so that we can understand the code after we’ve been away from it for a while or to understand someone else’s code, it is *absolutely essential* that we know and exercise good programming practices. The first step in that direction is to take responsibility for your work. Add “comment statements” to the top of the program, with your name and date, and perhaps how you can be reached (phone number, INTERNET address, etc.) And be sure to add a line or two about what the program does — or is supposed to do! Your modified code might then look something like the following:

```
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* HELLO.FOR is a little program that gets us started.
*
*           originally written:  9/1/87   pld
*           last revised:      1/1/93   pld
*
*       write(*,*)'hello, world
*       end
```

Since I modified existing code to get the current version, I included a history of the revisions, and initialed the entries. This is a little silly for this program, since it’s strictly an example program. But the idea is important — if we develop good habits now, we’ll have them when we really need them!

The first lines of every computer code should contain the programmer’s name, the date, and the purpose of the code.

Save your work, quit the editor, and `fl` your program. If *your* screen looked exactly like *my* screen, then it contained an error that the compiler has found. There’s a message on the screen that looks something like:

```
hello.for(9)  :  error F2031:  closing quote missing
```

This informs you that there is an error in line 9 of the file `hello.for`, and what the error is. (Unfortunately, not all the error messages are as clear as this one.) The error number, F2031 in this case, usually doesn’t help much, but the message clearly tells me that I forgot the second quote. (All the error messages

are listed in the Microsoft FORTRAN Reference manual.) Now, I knew it was supposed to be there, honest! It was even there previously — look at the listing! Apparently, in making modifications to the code I accidentally deleted the quote mark. I always try to be careful, but even experienced programmers make silly mistakes. I'll have to invoke my editor, correct my mistake, and fl the program again.

(With many editors, you can invoke the FORTRAN compiler without leaving the editor itself. Simply hit the appropriate key — perhaps F5 — and the editor will save the file and instruct the operating system to compile and link the program. One of the advantages of “compiling within the editor” is that the editor will place the cursor on the line of computer code that produced the error, and the FORTRAN error message will be displayed. You can then edit the line appropriately, to correct your error. If you have several errors, you can move through the file, correcting them one at a time. Proceeding in this way, editors provide a valuable tool for catching simple coding errors. Unfortunately, they don't detect errors of logic!)

After successfully compiling the program, type:

```
DIR
```

This commands the computer to provide you with a list of all the files in the directory. The following files should be present, although not necessarily in this order:

```
HELLO.BAK
HELLO.OBJ
HELLO.EXE
HELLO.FOR
```

A new file, HELLO.BAK, has appeared. It is a copy of HELLO *as it appeared before the last editing session*; that is, it is a *backup copy* of the program. HELLO.FOR is the current version containing the changes made in the last editing session, and HELLO.OBJ and HELLO.EXE are the newly created object and executable files — the old files were overwritten by the compiler and linker. Now when you run the program, your efforts should be rewarded by a pleasant greeting from HELLO!

A Numerical Example

While HELLO has given us a good start, most of the work we want to do is of a

numerical nature, so let's write a simple program that *computes* something. Let's see, how about computing the average between two numbers that you type into the keyboard? Sounds good. You'll read in one number, then another, and then compute and display the average. The FORTRAN code might look something like this:

```

      Program AVERAGE
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* Program AVERAGE computes the average of two numbers
* typed in at the keyboard, and writes the answer.
*
*               originally written:  9/2/87   pld
*
      read(*,*)first
      read(*,*)second
      average = (first+second)/2
      write(*,*) average
      end

```

This code should compile without errors.

Let's briefly discuss how computers work internally. For each variable such as `first`, a small amount of memory is set aside. The *value* of the variable, to about 8 significant digits, is stored in this space. When two such numbers are multiplied, the result has 16 significant digits. But there's no room to store them — there's only room for 8! As calculations are performed the results are rounded so as to fit into the space allotted, introducing an error. After many calculations, the accuracy of the result can be severely degraded, as will be demonstrated in Exercise 1.2. This is called *round-off error*, and can be diminished by allocating more space for each variable, using a `DOUBLE PRECISION` statement.

Real variables should always be double precision.

Although “standard” FORTRAN allows default types for variables, such as single precision for any variable name not starting with `i`, `j`, `k`, `l`, `m`, or `n`, we have found this to lead to a lot of trouble. For example, plane waves are described by the function e^{ikx} . If we have occasion to write a computer code involving plane waves, it would only be natural for us to use the variable `k`. It would also be *good programming*, in that the computer program would

then closely match the mathematical equations. But unless otherwise stated, the variable k would be an integer! That is, the statement $k=1.7$ would *not* cause k to take on the value of 1.7 — rather, it would be the integer 1! Such errors happen to everyone, and are tough to find since the code will compile and run — it just won't give the correct answers! Rather than trusting our luck to catch these errors, we will avoid their occurrence in the first place by adhering to the following rule:

Declare *all* variables.

In the computer science/software engineering world, this is termed “strong typing,” and is an integral part of some programming languages, although not FORTRAN. With the options specified in Appendix A, `f1` will help you abide by this edict by issuing a WARNING whenever an undeclared variable is encountered. It is left to you, however, to respond to the warning. (As programs are compiled, various ERRORS and WARNINGS might be issued. Corrective action to remove *all* errors and warnings should be taken before any attempt is made to execute the program.)

With these thoughts in mind, our computer code now looks like

```

Program AVERAGE
*****
* Paul L. DeVries, Department of Physics, Miami University
*
* Program AVERAGE computes the average of two numbers
* typed in at the keyboard, and writes the answer.
*
*           originally written:  9/2/87   pld
*           last revised:      1/1/93   pld
*
* Declarations
*
      DOUBLE PRECISION first, second, average
      read(*,*)first
      read(*,*)second

      average = (first+second)/2.d0

      write(*,*) average
end
! To ensure that the
! highest precision
! is maintained,
! constants should
! be specified in
! the "D" format.
```

Use your editor to create a file named `average.for` containing these lines of code, and `f1` to create an executable file. If you now execute the program, by typing `average`, you will receive a mild surprise: *nothing happens!* Well, not exactly nothing — the computer is *waiting* for you to type in the first number! Herein lies a lesson:

Always write a *prompting message* to the screen before attempting to read from the keyboard.

It doesn't have to be much, and it doesn't have to be fancy, but it should be there. A simple

```
write(*,*)'Hey bub, type in a number!'
```

is enough, but you need something. Likewise, the output should contain some identifying information so that you know what it is. It's common for programs to begin as little exercises, only to grow in size and complexity as time goes by. Today, you know exactly what the output means; tomorrow, you think you know; and three weeks from next Tuesday you will have forgotten that you wrote the program. (But, of course, your authorship will be established when you look at the comments at the head of the file.)



Output should always be self-descriptive.

EXERCISE 1.1

Modify `average.for` by adding prompts before the input statements and rewrite output statements so that they are self-descriptive. Then change the program so that the number of addends, `NUMADD`, is arbitrary and is obtained when you run the program.

Code Fragments

Throughout this text are listed examples of code and fragments of programs, which are intended to help you develop your computational physics skills. To save you the drudgery of entering these fragments, and to avoid the probable errors associated with their entry, many of these fragments are included on

the distribution diskette found in the back of the book. Rarely are these fragments complete or ready-to-run — they're only guides, but they will (hopefully) propel you in a constructive direction. Exercises having a code fragment on the disk are indicated by a  in the margin. The next exercise, for example, has a  in the margin, indicating that a fragment pertaining to that exercise exists on the diskette. Being associated with Exercise 1.2, this fragment is stored in the file 1.2. Most of the exercises *do not* have code fragments, simply because they can be successfully completed by modifying the programs written for previous exercises. Exercise 1.2 is something of a rarity, in that the entire code is included. All the software distributed with this textbook, including these fragments, is discussed in Appendix A.

Let's return to the issue of double precision variables, and demonstrate the degradation due to round-off error that can occur if they are not used. Consider the simple program

```

Program TWO
-----
* Paul L. DeVries, Department of Physics, Miami University
*
* Program TWO demonstrates the value of DOUBLE PRECISION
* variables.
*
*                               last revised:  1/1/93   pld
*
* Declarations
*
*       real x
*       double precision y
*       integer i
*
*   If we start with one, and add one-millionth a million
*   times, what do we get?
*
*       x = 1.
*       y = 1.d0
*       do i = 1, 1000000
*           x = x + 0.000001
*           y = y + 0.000001d0
*       end do
*       write(*,*) ' Which is closer to two, ',x,' or ',y,'?'
*       end

```

Its operation is obvious, but its result is rather startling.

EXERCISE 1.2

Run Program TWO. Are you convinced that DOUBLE PRECISION variables are important?

We should emphasize that we have not actually *eliminated* the error by using double precision variables, we have simply made it smaller by several orders of magnitude. There are other sources of error, due to the approximations being made and the numerical methods being used, that have nothing to do with round-off error, and are not affected by the type declaration. For example, if we were to use x as an approximation for $\sin x$, we would introduce a substantial *truncation error* into the solution, whether the variable x is declared double precision or not! Errors can also be introduced (or existing errors magnified) by the particular algorithm being used, so that the *stability* of the algorithm is crucial. The consistent use of double precision variables effectively reduces the round-off error to a level that can usually be ignored, allowing us to focus our attention on other sources of error in the problem.

A Brief Guide to Good Programming

When asked what are the most important characteristics of a good computer program, many people — particularly novices — will say speed and efficiency, or maybe reliability or accuracy. Certainly these are desirable characteristics — reliability, in particular, is an extremely desirable virtue. A program that sometimes works, and sometimes doesn't, isn't of much value. But a program that is not as fast or as efficient or as accurate can still be of use, if we understand the limits of the program. Don't be concerned with *efficiency* — it's infinitely preferable to have a slow, fat code that works than to have a fast, lean code that doesn't! And as for *speed* — the only “time” that really matters is how long it takes you to solve the problem, not how long it takes the program to execute. The reason the computer program was written in the first place was to solve a particular problem, and if it solves that problem — within known and understood limitations — then the program must be deemed a success. How best can we achieve that success?

I have come to the conclusion that the single most important characteristic of any computer program in computational physics is *clarity*. If it is not clear what the program is attempting to do, then it probably won't do it. If it's not clear what methods and algorithms are being used, then it's virtually impossible to know if it's working correctly. If it's not clear what the variables represent, we can't determine that the assignments and operations are valid. If the program is not clear, its value is essentially zero.

On the other hand, if a program is written to be as clear as possible, then we are more likely to understand what it's intended to do. If (and when) errors of logic are made, those errors are more easily recognized because we have a firm understanding of what the code was designed to do. Simple errors, such as mistaking one variable for another, entry errors, and so on, become much less likely to be made and much easier to detect. Modifying the code or substituting a new subroutine for an old is made simpler if we clearly understand how each subroutine works and how it relates to the rest of the program. Our goal is to “solve” the given problem, but the path to that goal is made much easier when we strive to write our programs with clarity.

Begin by considering the process of writing a program as a whole. In the early days of computing there was generally a rush to enter the code and compile it, often before the problem was totally understood. Not surprisingly, this resulted in much wasted effort — as the problem became better understood, the program had to be modified or even totally rewritten. Today, the entire discipline of “software engineering” has arisen to provide mechanisms by which programs can be reliably written in an efficient manner. The programs we will develop are generally not so long or involved that all the formal rules of software development need be imposed — but that doesn't mean we should proceed haphazardly, either.

Think about the problem, not the program.

A standard approach to any new and difficult problem is to break the original problem into more manageably sized pieces, or “chunks.” This gives us a better understanding of the total problem, since we now see it as a sequence of smaller steps, and also provides a clue as to how to proceed — “chunk” it again! Ultimately, the problems become small enough that we can solve them individually, and ultimately build a solution to the original problem.

When applied to computational problems, this problem-solving strategy is called *top-down design* and *structured programming*. This strategy enhances our understanding of the problem while simultaneously clarifying the steps necessary for a computational solution. In practice, top-down design and structured programming are accomplished by creating a hierarchy of steps leading to the solution, as depicted in Figure 1.1. At the highest level, the problem is broken into a few logical pieces. Then each of these pieces is broken into its logical pieces, and so on. At each level of the hierarchy, the steps become smaller, more clearly defined, and more detailed. At some point, the individual steps are the appropriate size for a SUBROUTINE or FUNCTION. There are no hard-and-fast rules on how small these units should be — the logical connectedness of the unit is far more important than the physical length —

but subroutines exceeding a hundred lines or so of code are probably rare. Functions, being somewhat more limited in scope, are usually even smaller. Top-down design doesn't stop at this level, however. Subroutines and functions themselves can be logically broken into separate steps, so that top-down design extends all the way down to logical steps within a program unit.

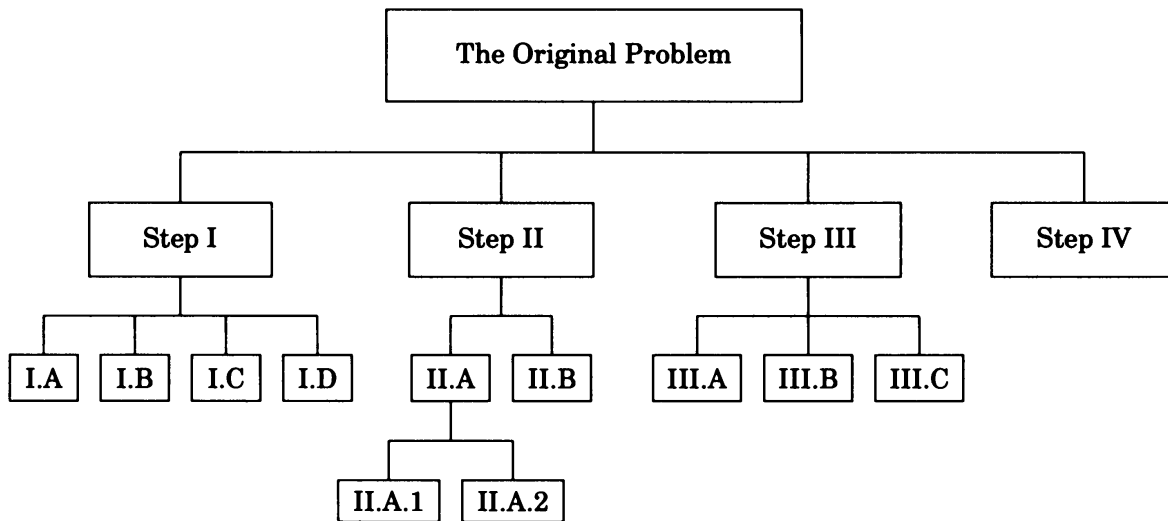


FIGURE 1.1 An example of the hierarchy resulting from top-down program design.

Clearly, a lot of effort is expended in this design phase. The payoff, at least partially, is that the programming is now very easy — the individual steps at the lowest, most detailed level of the design are implemented in computer code, and we can work our way up through the design hierarchy. Since the function of each piece is clearly defined, the actual programming should be very straightforward. With the top-down programming design, a hierarchy was developed with each level of refinement at a lower, more detailed level than the previous one. As we actually write the code, the process is reversed — that is, each discrete unit of code is written separately, and then combined with other units as the next higher level is written.

Alternatively, one can begin the programming in conjunction with the top-down design phase. That is, as the top levels of the program are being designed, the associated code can be written, calling as-yet unwritten subroutines (and functions) from the lower levels. Since the specific requirements of the lower-level subroutines will not be known initially, this code will normally be incomplete — these *stubs*, however, clearly reflect the design of the program and the specific areas where further development is required. The most important factor in either case is that the coding is secondary to the design of the program itself, which is accomplished by approaching the total

problem one step at a time, and successively refining those steps.

Design down.

As we write the computer code for these discrete steps, we are naturally led to *structured programming*. Subroutines at the lower levels of the hierarchy, for example, contain logical blocks of code corresponding to the lowest steps of the design hierarchy. Subroutines at the higher levels consist of calls to subroutines and functions that lie below them in the hierarchy. We can further enhance the structure of the program by insisting on linear program flow, so that control within a subroutine (or within a block of code) proceeds from the top to the bottom in a clear and predictable fashion. GOTO statements and statement labels should be used sparingly, and there should *never* be jumps into or out of a block of code. The IF-THEN-ELSE construct is a major asset in establishing such control. To assist in readability, capitalization is used to help identify control structures such as IF-THEN-ELSE and DO-loops. We also use indentation to provide a visual clue to the “level” of the code and to separate different blocks of code.

As you develop the actual code, constantly be aware of the need for clarity. There’s rarely a single line or block of code that works — usually, there are many different ways a specific task can be implemented. When confronted with a choice, choose the one that leads to the clearest, most readable code. Avoid *cleverness* in your coding — what looks clever today might look incomprehensible tomorrow. And don’t worry about *efficiency* — modern compilers are very good at producing efficient code. Your attempts to optimize the code might actually thwart the compiler’s efforts. The primary technique for improving efficiency is changing algorithms, not rewriting the code. (Unless you know *what* needs to be rewritten, and *how* to improve it, your efforts will likely be wasted, anyway.)

Overall program clarity is further promoted by appropriate documentation, beginning with a clear statement of what the program, subroutine, or function is *supposed* to be doing. It may happen that the code doesn’t actually perform as intended, but if the intention isn’t clearly stated, this discrepancy is difficult to recognize. The in-line documentation should then describe the method or algorithm being used to carry out that intention, at the level of detail appropriate for that subprogram in the design hierarchy. That is, a subroutine at the highest level should describe the sequence of major steps that it accomplishes; a description of how each of these steps is accomplished is contained within the subroutines that accomplish them. This description need not be — and in fact, should not be — a highly detailed narrative. Rather,

the requirement is that the description be *clear* and *precise*. If the Newton–Raphson method is being used, then the documentation should say so! And if a particular reference was used to develop the implementation of the Newton–Raphson method, then the reference should be cited!

We’ve already argued that each program, subroutine, and function should contain information describing the intended function of the subprogram, the author, and a revision history. We’ve now added to that list a statement of the method used and references to the method. Furthermore, the input and output of the program unit should be described, and the meaning of all variables. All this information should reside at the beginning of the program unit. Then within the unit, corresponding to each block of structured code, there should be a brief statement explaining the function of that block of code. Always, of course, the purpose of this documentation is clarity — obvious remarks don’t contribute to that goal. As an example, there are many instances in which we might want to calculate the value of a function at different points in space. A fragment of code to do that might look like

```
* Loop over "i" from 1 to 10
  DO i= 1, 10
    ...
  END DO
```

The comment statement in this fragment does not contribute to clarity. To the extent that it interferes with reading the code, it actually *detracts* from clarity. A better comment might be

```
* Loop over all points on the grid
```

But it still doesn’t tell us *what* is being done! A truly useful comment should *add* something to our understanding of the program, and not simply be a rewording of the code. An appropriate comment for this case might be

```
* Calculate the electrostatic potential at
*   all points on the spatial grid.
```

This comment succinctly describes what the block of code will be doing, and significantly contributes to the clarity of the program. Good documentation is not glamorous, but it’s not difficult, either.

Another way to enhance the clarity of a program is through the choice of variable names. Imagine trying to read and understand a computer code calculating thermodynamic quantities in which the variable `x34` appears. The name `x34` simply does not convey much information, except perhaps that it’s

the 34th variable. Far better to call it pressure, if that's what it is. (Within standard FORTRAN 77, this variable name is too long. However, as noted, Microsoft (and many other) compilers will accept the longer name. In FORTRAN 90, the limit is 31 alphanumeric characters.) Using well-named variables not only helps with keeping track of the variables themselves, it helps make clear the relations between them. For example, the line

```
x19=x37*y12
```

doesn't say much. Even a comment might not really help:

```
* Calculation of force from Newton's second law
x19=x37*y12
```

But when well-chosen names are used, the meaning of the variables and the relation between them become much clearer:

```
Force = Mass * Acceleration
```

Because the meaning of the FORTRAN statement is now obvious, we don't need the comment statement at all! This is an example of *self-documenting code*, so clear in what is being done that no further documentation is needed. It's also an example of the extreme clarity that we should strive for in all our programming efforts. When good names are combined with good documentation, the results are programs that are easy to read and to understand, and to test and debug — programs that work better, are written in a shorter amount of total time, and provide the solution to your problems in a timely manner.

A result of top-down design and structured programming is that the subroutines can easily be made self-contained. Such modularity is a definite advantage when testing and debugging the program, and makes it easier to maintain and modify at a later time. During the process of refinement, the purpose and function of each subroutine has been well defined. Comments should be included in the subroutine to record this purpose and function, to specify clearly the required input and output, and to describe precisely how the routine performs its function. Note that this has the effect of *hiding* much detailed information from the rest of the program. That is, this subroutine was designed to perform some particular task. The rest of the program doesn't need to know how it's done, only the subroutine's required input and expected output. If, for some reason, *we* want to know, then the information is there, and the program as a whole is not overburdened by a lot of unnecessary detail. Furthermore, if at some later time we want to replace this subroutine, we will then have all the information to know exactly what must be replaced.

Debugging and Testing

When the programming guidelines we've discussed are utilized, the resulting programs are often nearly error-free. Still, producing a totally error-free program on the first attempt is relatively rare. The process of finding and removing errors is called *debugging*. To some, debugging is nearly an art form — perhaps even a black art — yet it's a task that's amenable to a systematic approach.

The entire debugging process is greatly facilitated by our top-down, structured programming style, which produces discrete, well-defined units of code. The first step is to compile these units individually. Common errors, such as misspelling variable names and having unmatched parenthesis, are easily detected and corrected. With the specified options to `f1`, undeclared variables will also be detected and can be corrected, and variables that have been declared but are unused will be reported. All errors and warnings generated by the compiler should be removed before moving on to the next phase of the process.

After a subroutine (or function) has been successfully compiled, it should be tested before being integrated into the larger project. These tests will typically include several specific examples, and comparing the results to those obtained by hand or from some other source. It's tempting to keep these tests as simple as possible, but that rarely exercises the code sufficiently. For example, a code that works perfectly with an input parameter of $x = 1.0, 2.0$, or 3.0 might fail for $x = 1.23$. Remember, you are trying to see if it will *fail*, not if it will succeed. Another item that should always be tested is the behavior of the code at the ends — the first time through a loop, or the last, are often where errors occur.

Another common problem that is easily addressed at this level is one of data validity. Perhaps you've written a `FUNCTION` to evaluate one of the special functions that often arise in physics, such as the Legendre polynomial, a function defined on the interval $-1 \leq x \leq 1$. What does the code do if the argument $x = 1.7$ is passed to it? Obviously, there is an error somewhere for this to have happened. But that error is *compounded* if this `FUNCTION` doesn't recognize the error. All functions and subroutines should check that the input variables are reasonable, and if they're not, they should write an "error message" stating the nature of the problem and then terminate the execution of the program.

This is a good point at which to *rethink* what's been done. As the program was designed, an understanding of the problem, and of its solution, was developed. Now, a step in that solution has been successfully implemented,

and your understanding is even deeper. This is the time to ask, *Is there a better way of doing the same thing?* With your fuller understanding of the problem, perhaps a clearer, more concise approach will present itself. In particular, complex, nested, and convoluted control structures should be reexamined — Is it *really* doing what you want? Is there a more direct way of doing it?

You should also think about *generalizing* the subroutine (or function). Yes, it was designed to perform a specific task, and you’ve determined that it does it. But could it be made more general, so that it could be used in other situations? If you needed to do a task *once*, you’ll probably need to do it *again!* Invest the extra effort now, and be rewarded later when you don’t have to duplicate your work. A little additional effort here, while the workings of the subroutine are fresh in your mind, can make subsequent programming projects much easier.

Once we have ascertained that the code produces correct results for typical input, thoroughly documented the intended purpose of the code and the methods used, included input verification, reexamined its workings and perhaps generalized it, the code can be marked “provisionally acceptable” and we can move on to the next subprogram. After all the subroutines and functions within one logical phase of the project are complete, we can test that phase. We note that the acceptance of a subroutine or program is *never* more than provisional. The more we use a particular piece of code, the more confident we become of it. However, there is always the chance that within it there lurks a bug, just waiting for an inopportune time to crawl out.

A Cautionary Note

From time to time, we all make near catastrophic mistakes. For example, it’s entirely possible — even *easy* — to tell the computer to delete all your work. Clearly, you wouldn’t do this intentionally, but such accidents happen more often than you might think. Some editors can help — they maintain copies of your work so that you can **UnDo** your mistakes or **Restore** your work in case of accidents. Find out if your editor does this, and if so, learn how to use it! There are also “programmer’s tools” that allow you to recover a “deleted” file. As a final resort, make frequent backups of your work so that you never lose more than the work since the last backup. For example, when entering a lot of program code, or doing a lot of debugging, make a backup every half-hour or so. Then, if the unspeakable happens, you’ve only lost a few minutes of work. Plan now to use one or more of these “safety nets.” A little time invested now can save you an immense amount of time and frustration later on.

Elementary Computer Graphics

It's often the case that a sketch of a problem helps to solve and understand the problem. In fact, *visualization* is an important topic in computational physics today — computers can perform calculations much faster than we can sift through paper printout to understand them, and so we are asking the computer to present those results in a form more appropriate for human consumption, e.g., graphically. Our present interest is very simple — instead of using graph paper and pencil to produce our sketches by hand, let's have the computer do it for us.

The FORTRAN language doesn't contain graphics commands itself. However, graphics are so important that libraries containing graphical subroutines have been written, so that graphics can be done from within FORTRAN programs. In particular, Microsoft distributes such a library with its FORTRAN compiler, Version 5.0 or later. But creating high-quality graphics is quite difficult, even with all the tools. Our emphasis will be in producing relatively simple graphs, for our own use, rather than in creating plots to be included in textbooks, for example. With this in mind, we have chosen to use only a few of the graphics subroutines available to us. The results will be simplistic, in comparison to what some scientific graphics packages can produce, but entirely adequate for our purposes. We have even simplified the access to the library, so that simple plots can be produced with virtually no knowledge of the Microsoft graphics library. (See Appendix B for more information about graphics.)

There are two subroutines that must be called for every plot:

gINIT ... initializes the graphics package and prepares the computer and the graphical output device for producing a drawing. It *must* be the first graphics subroutine called.

gEND ... releases the graphics package, and returns the computer to its standard configuration. This should be the last call made, after you have finished viewing the graph you created — the last action taken is to clear the screen.

After **gINIT** has been called, the entire screen is available for drawing. Where your graph will be displayed is referred to as the *viewport*: as this discussion might suggest, it's possible to place the viewport anywhere on the physical display device — at this time, the viewport covers the entire screen. To change the location (or size) of the viewport, use

VIEWPORT(X1, Y1, X2, Y2) ... locates the viewport on the physical display

device. Scaled coordinates are used: the lower left side of the display is the point (0,0), and the upper right side is (1,1).

Before the data can be displayed, you must inform the graphics library of the range of your data. That is, you must map the lower and upper limits of your data onto the physical screen, e.g., the viewport. This is called a *window* on your data. After we work through the jargon, the *viewport* and *window* concepts make it very easy for us to design our graphs. Remember, the coordinates of the viewport relate to the physical display device while the coordinates of the window relate to the particular data being displayed. So, we have the command WINDOW,

WINDOW(X1, Y1, X2, Y2) ... maps data in the range $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$ onto the viewport.

We're now ready to consider the commands that actually produce the plot. After the viewport and the window have been set, we're ready to draw a line.

LINE(X1, Y1, X2, Y2) ... draws a line from (x_1, y_1) to (x_2, y_2) .

The parameters in the call to LINE are simply the x and y values of your data in their natural units, as declared to the graphics package by WINDOW. Both endpoints of the line are specified.

This might all seem a little confusing — let's look at a simple program to demonstrate these routines:

```

      Program DEMO
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This little program demonstrates the use of some of the
* elementary graphics commands available.
*                               January 1, 1993
*
*       double precision x1,y1,x2,y2
*
* Start the graphics package, and initialize WINDOW.
*
*       call gINIT
*
* the data range is      -10 < x < 10,
*                        0 < y < 1.

```

```

*
    x1 = -10.d0
    x2 = +10.d0
    y1 = 0.d0
    y2 = 1.d0

    call WINDOW( x1, y1, x2, y2 )

    call LINE( -10.d0, .5d0, 10.d0, .5d0 )
    call LINE( 0.d0, 0.d0, 0.d0, 1.d0 )

    call gEND

END

```

This program should produce a “+” on your computer screen.

These routines must always be called with double precision arguments. The preferred approach is to use variables as arguments, as WINDOW was called. If called with constants, the arguments MUST be specified in a D format, as LINE was called. Failure to provide double precision arguments will cause unpredictable results.

EXERCISE 1.3

Verify that DEMO works as advertised.

Let’s work through something a little more complicated: plotting a sine curve, on the domain $0 \leq x \leq 10$. To approximate the continuous curve, we’ll draw several straight-line segments — as the number of line segments increases, the graph will look more and more like a continuous curve. Let’s use 50 line segments, just to see what it looks like. It would also be nice if the axes were drawn, and perhaps some tickmarks included in the plot. Finally, let’s use a viewport smaller than the entire screen, and locate it in the upper middle of the physical screen. The appropriate code might then look like the following:

```

          Program SINE
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* A quick example:  drawing a sine curve.
*
                                January 1, 1993

```

```

*
    double precision x1,y1,x2,y2
    integer i
*
* Start the graphics package, and initialize WINDOW.
*
    call gINIT
*
* Put the viewport in the upper middle of the screen.
*
    x1 = 0.25d0
    x2 = 0.75d0
    y1 = 0.40d0
    y2 = 0.90d0
    call VIEWPORT( x1, y1, x2, y2 )
*
* The data range is      0 < x < 10,
*                        -1 < y < 1.
*
    x1 = 0.d0
    x2 = +10.d0
    y1 = -1.d0
    y2 = 1.d0
    call WINDOW( x1, y1, x2, y2 )
*
* Draw x-axis
*
    call LINE( x1, 0.d0, x2, 0.d0 )
*
* Put tickmarks on x-axis
*
    y1 = -0.04d0 ! about 1/50 of the y range
    y2 = 0.00d0
    DO i = 1, 10
        x1 = dble(i)
        call LINE( x1, y1, x1, y2 )
    END DO
*
* Draw y-axis
*
    call LINE( x1, y1, x1, y2 )
*
* Draw sine curve
*

```

```

DO  i = 1, 50
    x1 = dble(i-1)*0.2d0
    y1 = sin(x1)
    x2 = dble( i )*0.2d0
    y2 = sin(x2)
    call LINE( x1, y1, x2, y2 )
END DO

call gEND
END

```

This code should produce the line drawing in Figure 1.2. For us, graphics are an aid to our understanding, so that simple drawings such as this are entirely adequate. We should, however, add some labeling to the figure. Text can be written with a `WRITE` statement, but we need to position the label appropriately. To position the cursor, we use the command

CURSOR(row, column) ... which moves the cursor to the designated row and column. (The row numbering is from top to bottom.) *row* and *column* are *integer* variables.

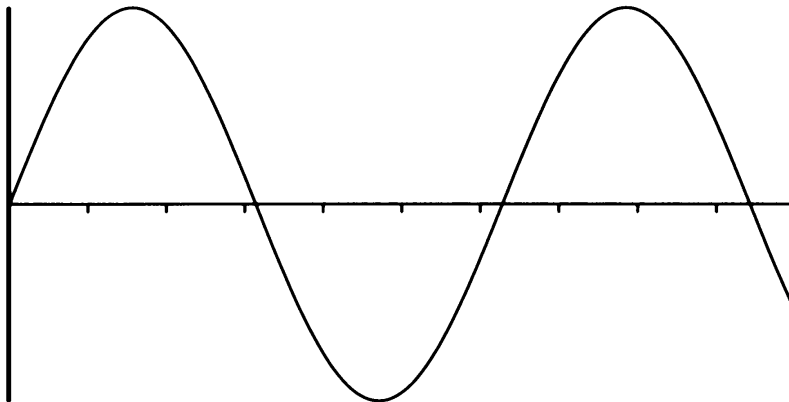


FIGURE 1.2 Plot of a sine curve, $0 < x < 10$.

EXERCISE 1.4

Reproduce Figure 1.2, with the appropriate legend added.

Of course, there's no restriction to plotting only one curve on a figure!

EXERCISE 1.5

Plot both the sine and cosine curves in a single figure.

And in Living Color!

The addition of color greatly enhances the quality of graphic presentations. If your computer is equipped with a color display, you can determine the number of colors available to you with

NOC(NUMBER) ... which returns the number of colors available in the integer number.

To change the color being used, we can use the subroutine

COLOR(INDEX) ... sets the current color to the one indexed by INDEX. All subsequent lines will be drawn in this color until changed by another call to COLOR. Under normal circumstances, INDEX = 0 is black, while the highest indexed color is white.

In general, we'll find color very useful. For example, we can distinguish between two curves drawn on the same graph by drawing them with different colors. If the indices of the two desired colors are "1" and "2," then we might have

```

...
*
*  draw first curve
*
      call color(1)
      DO i = ...
        ...
      END DO
*
*  draw second curve
*
      call color(2)
      DO i = ...
        ...
      END DO
...

```

EXERCISE 1.6

Repeat the previous exercise, drawing the sine curve in one color and the cosine curve in another.

Color can also be used to draw attention to the plot, and to add an

aesthetic element to it. An added background color can be visually pleasing, while emphasizing the presence of the plot itself. In combination with `CURSOR` to label the graph and provide textual information, the result can be impressive. A rectangular box can be filled with a color using

FILL(X1, Y1, X2, Y2) ... fills the box defined by the corners (x_1, y_1) and (x_2, y_2) with the current color.

For example, a background color can easily be added to your sine plot:

```

        call WINDOW ( x1, y1, x2, y2 )
        call NOC ( NUMBER )
*
*   Fill the background with the color with INDEX = 2
*
        call COLOR(2)
        call FILL ( x1, y1, x2, y2 )
*
*   Restore color index to white
*
        call COLOR ( NUMBER-1 )
        ...

```

EXERCISE 1.7

Add a background color to your sine/cosine drawing program. You will want to experiment with different indices to find the color most appealing to you.

Classic Curves

We feel compelled to comment that while graphics are extraordinarily useful, they're also a lot of fun. And that's great — learning should be an enjoyable process, else it's unlikely to be successful! The ease with which we can generate curves and figures also inspires us to *explore* the various possibilities. Of course, we're not the first to tread this path — generations of mathematicians have investigated various functions and curves. Let's apply the modern capabilities of computers, and graphing, to some of the classic analytic work, for the sole purpose of enjoying and appreciating its elegance. Table 1.1 presents some of the possibilities.

TABLE 1.1 Classic Curves in the Plane

Bifolium	$(x^2 + y^2)^2 = ax^2y$ $r = a \sin \theta \cos^2 \theta$
Cissoid of Diocles	$y^2(a - x) = x^3$ $r = a \sin \theta \tan \theta$
Cochleoid	$(x^2 + y^2) \tan^{-1}(y/x) = ay$ $r\theta = a \sin \theta$
Conchoid of Nicomedes	$(y - a)^2(x^2 + y^2) = b^2y^2$ $r = a \csc \theta \pm b$
Deltoid	$x = 2a \cos \phi + a \cos 2\phi$ $y = 2a \sin \phi - a \sin 2\phi$
Evolute of ellipse	$(ax)^{2/3} + (by)^{2/3} = (a^2 - b^2)^{2/3}$ $x = \frac{a^2 - b^2}{a} \cos^3 \phi$ $y = \frac{a^2 - b^2}{b} \sin^3 \phi$
Folium of Descartes	$x^3 + y^3 - 3axy = 0$ $r = \frac{3a \sin \theta \cos \theta}{\cos^3 \theta + \sin^3 \theta}$
Hypocycloid with four cusps	$x^{2/3} + y^{2/3} = a^{2/3}$ $x = a \cos^3 \phi, y = a \sin^3 \phi$
Involute of a circle	$x = a \cos \phi + a\phi \sin \phi$ $y = a \sin \phi - a\phi \cos \phi$
Lemniscate of Bernoulli	$(x^2 + y^2)^2 = a^2(x^2 - y^2)$ $r^2 = a^2 \cos 2\theta$
Limacon of Pascal	$(x^2 + y^2 - ax)^2 = b^2(x^2 + y^2)$ $r = b + a \cos \theta$
Nephroid	$x = a(3 \cos \phi - \cos 3\phi)$ $y = a(3 \sin \phi - \sin 3\phi)$
Ovals of Cassini	$(x^2 + y^2 + b^2)^2 - 4b^2x^2 = k^4$ $r^4 + b^4 - 2r^2b^2 \cos 2\theta = k^4$
Logarithmic Spiral	$r = e^{a\theta}$
Parabolic Spiral	$(r - a)^2 = 4ak\theta$
Spiral of Archimedes	$r = a\theta$
Spiral of Galileo	$r = a\theta^2$
Strophoid	$y^2 = x^2 \frac{a - x}{a + x}$ $r = a \cos 2\theta \sec \theta$

Three-leaved rose

$$r = a \sin 3\theta$$

Tractrix

$$x = a \operatorname{sech}^{-1} y/x - \sqrt{a^2 - y^2}$$

Witch of Agnesi

$$y = \frac{8a^3}{x^2 + 4a^2}$$

$$x = 2a \cot \phi, y = a(1 - \cos 2\phi)$$

Note that some of these curves are more easily described in one coordinate system than another. Other curves are most easily expressed parametrically: both x and y are given in terms of the parameter ϕ .

EXERCISE 1.8

Plot one (or more) of these classic figures. Many of these curves are of physical and/or historic interest — do a little reference work, and see what you can find about the curve you chose.

The availability of computer graphics has encouraged the exploration of functions which otherwise would never have been considered. For example, Professor Fey of the University of Southern Mississippi suggests the function

$$r = e^{\cos \theta} - 2 \cos 4\theta + \sin^5 (\theta/12), \quad (1.1)$$

an eminently humane method of butterfly collecting.

EXERCISE 1.9

Plot Fey's function on the domain $0 \leq \theta \leq 24\pi$. The artistic aspect of the curve is enhanced if it's rotated 90° by using the transformation

$$x = r \cos(\theta + \pi/2),$$

$$y = r \sin(\theta + \pi/2).$$

Monster Curves

The figures we've been discussing are familiar to all those with a background in real analysis, and other "ordinary" types of mathematics. But mathematicians are a funny group — around the turn of the century, they began exploring some very unusual "curves." Imagine, if you will, a curve that is everywhere continuous, but nowhere differentiable! A real *monster*, wouldn't

you say? Yet the likes of Hilbert, Peano, and Sierpinski investigated these monsters — not a lightweight group, to say the least.

One particularly curious figure was due to Helge von Koch in 1904. As with many of these figures, the easiest way to specify the curve is to describe how to construct it. And perhaps the easiest way to describe it is with *production rules*. (This is the “modern” way of doing things — Helge didn’t know about these.)

Let’s imagine that you’re instructing an incredibly stupid machine to draw the figure, so we want to keep the instructions as simple as possible. To describe the von Koch curve, we need only four instructions: F , to go forward one step; $+$, to turn to the right by 60° ; $-$, to turn to the left by 60° ; and T , to reduce the size of the step by a factor of one-third. To construct the von Koch curve, we begin with an equilateral triangle with sides of unit length. The instructions to draw this triangle are simply

$$F + +F + +F + +. \quad (1.2)$$

To produce a new figure, we follow two rules: first, add a T to the beginning of the instruction list; and second, replace F by a new set of instructions:

$$F \rightarrow F - F + +F - F. \quad (1.3)$$

That is, every time an F appears in the original set of instructions, it is replaced by $F - F + +F - F$. If we follow these rules, beginning with the original description of the figure, we produce a new list of instructions:

$$TF - F + +F - F + +F - F + +F - F + +F - F + +F - F + +. \quad (1.4)$$

The first such figures are presented in Figure 1.3. To obtain the ultimate figure, as depicted in Figure 1.4, simply repeat this process an infinite number of times!

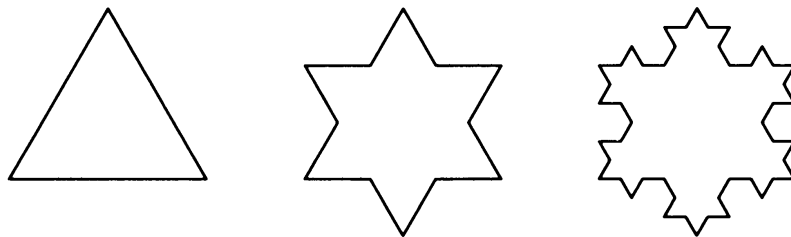


FIGURE 1.3 The first three steps of the von Koch construction.

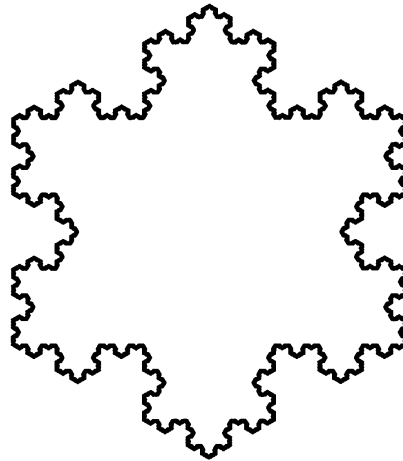


FIGURE 1.4 The von Koch curve, a.k.a. the Snowflake.

How long is the curve obtained? A “classical” shape, like a circle, has a length that can be approached by taking smaller and smaller chords, larger and larger n -polygon approximations to the circle. It’s an “ordinary,” finite length line on a two-dimensional surface. But the von Koch curve is not so ordinary. The length of the original curve was three units, but then each side was replaced by four line segments of one-third the original length, so that its length was increased by four-thirds. In fact, *at every iteration*, the length of the curve is increased by four-thirds. So, the length of the curve after an infinite number of iterations is infinite! Actually, the length of the curve between any two points on it is infinite, as well. And yet it’s all contained within a circle drawn around the original equilateral triangle! Not a typical figure.

We can invent other curves as well, with the addition of some new instructions. Let’s add R , to turn right 90° , L to turn left 90° , and Q to reduce the step length by a factor of a quarter. A square can then be drawn with the instructions

$$FRFRFRFR. \quad (1.5)$$

An interesting figure can then be produced by first appending Q to the front of the list, and making the replacement

$$F \rightarrow FLFRFRFFLFLFRF. \quad (1.6)$$

EXERCISE 1.10

Draw the figure resulting from three iterations of these production rules, which we might call a variegated square. Note that while the

length of the perimeter continues to increase, the area bounded by the perimeter is a constant.

Should we care about such monsters? Do they describe anything in Nature? The definitive answer to that question is not yet available, but some interesting observations can be made about them, and their relation to the physical world.

For example, exactly how long is the coastline of Lake Erie? If you use a meter stick, you can measure a certain length, but you clearly have not measured the *entire* length because there are features smaller than a meter that were overlooked. So you measure again, this time with a half-meter stick, and get a new “length.” This second length is larger than the first, and so you might try again with a shorter stick, and so on. Now the question is: Do the lengths converge to some number. The answer is: Not always — the von Koch curve, for example.

Benoit Mandelbrot has spent the better part of the last thirty years looking at these questions. He has coined the word *fractal* to describe such geometries, because the curve is something more than a line, yet less than an area — its dimension is *fractional*.

Consider a line segment. If we divide it into N identical pieces, then each piece is described by a scale factor $r = 1/N$. Now consider a square: if it is divided into N identical pieces, each piece has linear dimensions $r = 1/\sqrt{N}$. And for a cube, $r = 1/\sqrt[3]{N}$. So, apparently, we have

$$r = \frac{1}{\sqrt[D]{N}}, \quad (1.7)$$

or

$$D = \frac{\log N}{\log(1/r)}, \quad (1.8)$$

which we’ll take as the definition of the fractal dimension. Back to the von Koch curve: in the construction, each line segment is divided into 4 pieces ($N = 4$), scaled down by a factor of 3 ($r = 1/3$). Thus

$$D = \frac{\log 4}{\log 3} \approx 1.2618 \dots \quad (1.9)$$

This says that somehow the curve is more than a simple line, but less than an area.

EXERCISE 1.11

What is the fractal dimension of the variegated square?

The Mandelbrot Set

Mandelbrot has done more than coin a word, of course. Scarcely a person this side of Katmandu has not heard of and seen an image of the Mandelbrot set, such as Figure 1.5. It has become the signature of an entirely new area of scientific investigation: chaotic dynamics. And yet it's "created" by an extraordinarily simple procedure — for some complex number c , we start with $z = 0$ and evaluate subsequent z 's by the iteration

$$z = z^2 + c. \quad (1.10)$$

If z remains finite, even after an infinite number of iterations, then the point c is a member of the Mandelbrot set. (It was a simple matter to construct the von Koch curve, too!)

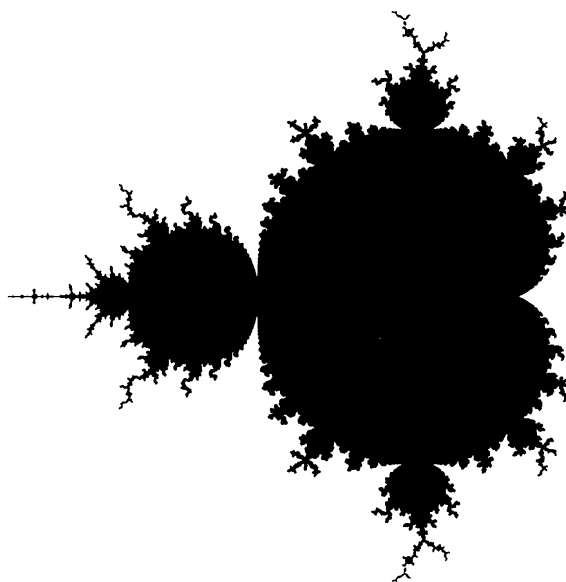


FIGURE 1.5 The Mandelbrot set.

We want to use our newfound graphics tools to help us visualize the Mandelbrot set. We will use the most basic, simpleminded, and incredibly slow method known to calculate the set: straightforward application of Equation (1.10). And even then, we'll only *approximate* the set — an infinite number of iterations can take a long time.

It turns out that if $|z|$ ever gets larger than 2, it will eventually become infinite. So we'll only iterate the equation a few times, say 30; if $|z|$ is still less than 2, then there's a good chance that c is a member of the set. The *computing* involved is rather obvious, and shouldn't be much trouble. The graphics, on the other hand, are a little trickier.

In our desire to achieve device-independence, we have "agreed" to forego knowledge about the actual graphics device. But in this instance, we need that knowledge. In particular, we want to fill the screen with relevant information, which means that we need to know the location of the individual pixels that make up the image. We can determine the maximum number of pixels of the screen by calling

MAXVIEW(NX, NY) ... returns the size of the physical display, in pixels. NX and NY are integer variables.

Thus (0,0) is one corner of the screen — actually, the upper left corner — and (NX,NY) is the opposite corner. To construct the image of Figure 1.5, we let $c = x + iy$ and considered the domain $-1.7 \leq x \leq 0.8$ and $-1.0 \leq y \leq 1.0$. That is, we mapped the domain of c onto the physical pixels of the screen. To illustrate, let's be somewhat conservative and consider only a limited portion of the screen, say, an 80×100 pixel region. Then the (x, y) coordinate of the (i, j) -th pixel is

$$x = -1.7 + i \frac{2.5}{100}, \quad (1.11)$$

and

$$y = -1.0 + j \frac{2.0}{80}. \quad (1.12)$$

As we cycle through the 80×100 array of pixels, we are considering different specific values of c . And for each c , we test to determine if it's a member of the Mandelbrot set by iterating Equation (1.10). The appropriate computer code to display the Mandelbrot set might then look like the following:

```

Program Mandelbrot
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This program computes and plots the Mandelbrot set,
* using a simpleminded and incredibly slow procedure.
*
*
* January 1, 1993
*
double precision x1, y1, x2, y2, x, y
complex*16 c, z, im
```

```

        integer i, j, NX, NY, counter, ix0, iy0
+         , Max_Iteration
        parameter( im = (0.d0,1.d0), Max_Iteration = 30 )
*
* Start the graphics package
*
        call gINIT
*
* Find the maximum allowed viewport size
*
        call MAXVIEW(nx,ny)
*
* When I know the code will work, I'll use more pixels.
* For right now, be conservative and use 100 x-pixels
* and 80 y-pixels, centered.
*
        ix0 = nx/2-50
        iy0 = ny/2-40
*
* Image to be computed in the region  $-1.7 < x < 0.8$ ,
*                                      $-1.0 < y < 1.0$ .
*
        x1 = -1.7d0
        x2 = 0.8d0
        y1 = -1.d0
        y2 = 1.d0
*
* Cycle over all the pixels on the screen to be included
* in the plot. Each pixel represents a particular complex
* number, c. Then determine, for this particular c, if
*  $z \rightarrow \text{infinity}$  as the iteration proceeds.
*
        DO i = 0, 100
            x = x1 + i * (x2-x1)/100.d0

            DO j = 0, 80
                y = y1 + j * (y2-y1)/80.d0

                C = X + im*Y
*
* Initialize z, and begin the iteration
*
                z = 0.d0
                counter = 0

```

```

100          z = z*z + c
            IF( abs(z) .lt. 2.d0 ) THEN
              counter = counter + 1
              if( counter .lt. Max_Iteration ) GOTO 100
*
*   If z is still < 2, and if counter = Max_Iteration,
*   call the point "c" a member of the Mandelbrot set:
*
              call pixel( ix0+i, iy0+j )
            ENDIF
          END DO
        END DO

        call gEND
      END

```

The variables *c*, *z*, and *im* have been declared `COMPLEX*16`, so that both their real and imaginary parts are double precision. If a pixel is found to be a member of the set, turning it on is accomplished by a call to the subroutine `pixel`,

PIXEL(I, J) ... turns on the *i*-th horizontal and *j*-th vertical pixel. This subroutine is device *dependent*, and should be used with some care.

As noted, with `MAXVIEW` we could have determined the maximum size of the computer screen and used all of the display, but we've chosen not to do so. That would cause the code to loop over *every* pixel on the screen, which can take quite a while to execute. Until we're sure of what we have, we'll keep the drawing region small.

EXERCISE 1.12

Produce a nice picture of the Mandelbrot set, following the suggestions presented here.

One of the really interesting things about fractals is their *self-similarity* — that is, as you look at the image closer and closer, you see similar shapes emerge. With the von Koch snowflake, this self-similarity was virtually exact. With the Mandelbrot set, the images are not exact duplicates of one another, but are certainly familiar.

EXERCISE 1.13

Pick a small section of the complex plane, where *something* is happening, and generate an image with enlarged magnification. For example,

you might investigate the region $-0.7 \leq x \leq -0.4$, $0.5 \leq y \leq 0.7$. By mapping these coordinates onto the same region of the screen used previously, you are effectively magnifying the image.

As you investigate the boundary of the Mandelbrot set at a finer scale, you should also increase the number of iterations used to determine membership in the set. Determining membership is an example of a fundamental limitation in computation — since we cannot in practice iterate an infinite number of times, we are always including a few points in our rendering that do not really belong there.

What sets fractals apart from other unusual mathematical objects is their visual presentation, as seen in numerous books and even television shows. There is even some debate as to whether “fractals” belong to mathematics, or to computer graphics. In either case, the images can certainly be striking. Their aesthetic appeal is particularly evident when they are rendered in color. In the current instance, we’ll use color to indicate the number of iterations a particular point c survived before it exceeded 2. That is, rather than denoting points *in* the Mandelbrot set, we’ll denote points *outside* the Mandelbrot set, with the color of the point indicating *how far* outside the set the point is. NOC can be used to determine the number of colors available, and the “membership” loop will be replaced by

```

        call NOC( number )
        ...
*
* Initialize z, and begin the iteration
*
        z = 0.d0
        counter = 0
100      z = z*z + c
        IF( abs(z) .lt. 2.d0 ) THEN
            counter = counter + 1
            if( counter .lt. Max_Iteration ) GOTO 100
        ELSE
*
* If z > 2, denote how far OUTSIDE the set
* the point "c" is:
*
            call COLOR( mod( counter, number ) )
            call PIXEL( ix0+i, iy0+j )
        ENDIF

```

Setting the color index in this way has the effect of cycling through all

the colors available on your particular computer. Since even on monochrome systems there are two “colors” — black and white — the images this produces can be quite interesting.

EXERCISE 1.14

Try this code to explore the Mandelbrot set, and see if you can find some particularly pleasing venues. Remember that as you “magnify” the image, you should also increase the iteration maximum beyond the thirty specified in the listing of the code.

These images can be almost addictive. Certainly, the images are very interesting. As you attempt greater magnification, and hence increase the iteration maximum, you will find that the calculation can be painfully slow. The problem is in fact in the iteration, particularly as it involves complex arithmetic. In general, we’re not particularly interested in efficiency. But in this case we can clearly identify where the code is inefficient, so that effort spent here to enhance its speed is well spent. Unfortunately, there is little that can be done within FORTRAN for this particular problem.

It is possible to gain a considerable improvement in speed *if* your computer is equipped with an 80x87 coprocessor, however. By writing in machine language, we can take advantage of the capabilities of the hardware. Such a subroutine has been written and is included in the software library distributed with this text. It can be used by replacing the previous code fragment with the following:

```
*
*****
*
*      Use ZIP_87 to perform the iteration.
*
*      Valid ONLY if your computer is equipped
*      with an 80x87 coprocessor!!!!!!
*
*****
*
*      call ZIP_87( c, Max_Iteration, counter )
*      IF( counter .lt. Max_Iteration) THEN
*
*      Magnitude of z IS greater than 2, "c" IS NOT a
*      member of the Mandelbrot set. Denote how far
*      OUTSIDE the set the point "c" is:
*
```

```

        call COLOR( mod( counter, number ) )
        call PIXEL( ix0+i, iy0+j )
    ENDIF

```

You should see a substantial increase in the speed of your Mandelbrot programs using ZIP_87, provided that your computer is equipped with the necessary hardware.

References

The ultimate reference for the FORTRAN language is the reference manual supplied by the manufacturer for the specific compiler that you're using. This manual will provide you with all the appropriate commands and their syntax. Such manuals are rarely useful in actually learning how to program, however. There are numerous books available to teach the elements of FORTRAN, but there are few that go beyond that. Three that we can recommend are

D.M. Etter, *Structured FORTRAN 77 For Engineers and Scientists*, Benjamin/Cummings, Menlo Park, 1987.

Michael Metcalf, *Effective FORTRAN 77*, Oxford University Press, Oxford, 1988.

Tim Ward and Eddie Bromhead, *FORTRAN and the Art of PC Programming*, John Wiley & Sons, New York, 1989.

Also of interest is the optimistic

Michael Metcalf and John Reid, *FORTRAN 8x Explained*, Oxford University Press, Oxford, 1987,

written before the delays that pushed the acceptance of the "new" standard into 1990.

The Mandelbrot set and the topic of fractals have captured the imagination of many of us. For the serious enthusiast, there's

Benoit B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman, New York, 1983.

Two of the finest books, both including many marvelous color photographs, are

H.-O. Peitgen and P. H. Richter, *The Beauty of Fractals*, Springer-Verlag, Berlin, 1986.

The Science of Fractal Images, edited by Heinz-Otto Peitgen and Dietmar Saupe, Springer-Verlag, Berlin, 1988.

Chapter 2:

Functions and Roots

A natural place for us to begin our discussion of computational physics is with a discussion of functions. After all, the formal theory of functions underlies virtually all of scientific theory, and their use is fundamental to any practical method of solving problems. We'll discuss some general properties, but always with an eye toward what is computationally applicable.

In particular, we'll discuss the problem of finding the roots of a function in one dimension. This is a relatively simple problem that arises quite frequently. Important in its own right, the problem provides us an opportunity to explore and illustrate the interplay among formal mathematics, numerical analysis, and computational physics. And we'll apply it to an interesting problem: the determination of quantum energy levels in a simple system.

Finding the Roots of a Function

We'll begin our exploration of computational physics by discussing one of the oldest of numerical problems: finding the x value for which a given function $f(x)$ is zero. This problem often appears as an intermediate step in the study of a larger problem, but is sometimes *the* problem of interest, as we'll find later in this chapter when we investigate a certain problem of quantum mechanics.

For low order polynomials, finding the zero of a function is a trivial problem: if the function is $f(x) = x - 3$, for example, the equation $x - 3 = 0$ is simply rearranged to read $x = 3$, which is the solution. Closed form solutions for the roots exist for quadratic, cubic, and quartic equations as well, although they become rather cumbersome to use. But no general solution exists for polynomials of fifth-order and higher! Many equations involving functions other than polynomials have no analytic solution at all.

So what we're really seeking is a method for solving for the root of a nonlinear equation. When expressed in this way, the problem seems anything

but trivial. To help focus our attention on the problem, let's consider a specific example: let's try to find the value of x for which $x = \cos x$. This problem is cast into a “zero of a function” problem simply by defining the function of interest to be

$$f(x) = \cos x - x. \quad (2.1)$$

Such transcendental equations are not (generally) solvable analytically.

The first thing we might try is to draw a figure, such as Figure 2.1, in which $\cos x$ and x are plotted. The root is simply the horizontal coordinate at which the two curves cross. The eye has no trouble finding this intersection, and the graph can easily be read to determine that the root lies near $x = 0.75$. But greater accuracy than this is hard to achieve by graphical methods. Furthermore, if there are a large number of roots to find, or if the function is not an easy one to plot, the effectiveness of this graphical method rapidly decreases — we have no choice but to attempt a solution by numerical means. What we would like to have is a reliable numerical method that will provide accurate results with a minimum of human intervention.

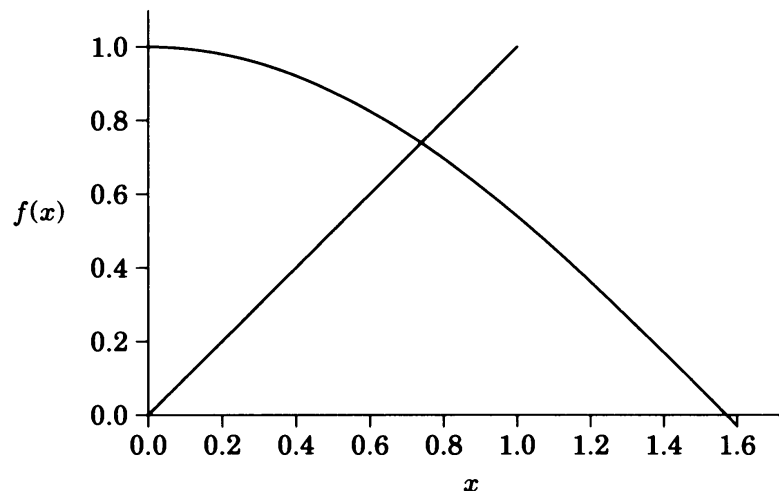


FIGURE 2.1 The functions x and $\cos x$.

From the figure, it's clear that a root lies between zero and $\pi/2$; that is, the root is *bracketed* between these two limits. We can improve our brackets by dividing the interval in half, and retaining the interval that contains the root. We can then check to see how small the bracket has become: if it is still too large, we halve the interval again! By doing this repeatedly, the upper and lower limits of the interval approach the root. Sounds like this just might work! Since the interval is halved at each step, the method is called *bisection*. The general construction of the computer code might be something like

```
< Main Program identification>
< declare variables needed here>
```

```

< initialize limits of bracket>
< call Root Finding subroutine >
< print result, and end>

```

*-----

```

< Subroutine identification>
< declaration of variables, etc.>
< prepare for looping:  set initial values, etc.>
< TOP of the loop>
    ...
< Body of the loop:  divide the interval in half,
                    determine which half contains the root,
                    redefine the limits of the bracket>
    ...
IF (bracket still too big) go to the TOP of the loop
    ...
< loop finished:
    put on finishing touches (if needed), and end>

```

These lines are referred to as *pseudocode* since they convey the intended content of the program but don't contain the actual executable statements. Let's first determine what the program should do, then worry about how to do it.

Think first, then code.

As indicated in the pseudocode, it's extremely important to use subroutines and functions to break the code into manageable pieces. This modularity aids immensely in the clarity and readability of the code — the main program simply becomes a list of calls to the various subroutines, so that the overall logic of the program is nearly transparent. Modularity also isolates one aspect of a problem from all the others: all the details of finding the root will be contained in the root-finding subroutine, in effect *hidden* from the rest of the program! This structure generally helps in developing and maintaining a program as well, in that a different approach to one aspect of the problem can be investigated by swapping subroutines, rather than totally rewriting the program.

After having a general outline of the entire program in pseudocode, we can now go back and expand upon its various components. In a large, complicated project, this refinement step will be performed several times, each time resulting in a more detailed description of the functioning of the code than the

previous one. In this process, the program practically “writes itself.” For this project, we might begin the refinement process by determining what needs to be passed to the subroutine, and naming those variables appropriately. For example, the subroutine should be given the limits of the bracketed interval: let’s name these variables `Left` and `Right`, for example, in the subroutine. (Note, however, that different names, such as `x_initial` and `x_final`, might be more appropriate in the main program.) In one sense, providing reasonable names for variables is simple stuff, easy to implement, and not very important. Not important, that is, until you try to remember (6 months from now) what a poorly named variable, like `x34b`, was supposed to mean!

Give your variables meaningful names, and declare them appropriately.

We also need to provide the subroutine with the name of the function. The function, or at least its name, could be defined within the subroutine, but then the subroutine would need to be changed in order to investigate a different function. One of the goals of writing modular code is to write it *once*, and *only once*! So we’ll declare the function as `EXTERNAL` in the main program, and pass its name to the subroutine. Oh, and we want the root passed back to the calling program! Our first refinement of the code might then look like

```

      < Main Program identification>
*
*   Type Declarations
*
      DOUBLE PRECISION x_initial, x_final, Root, FofX
*
      External FofX      ! The function f(x) will be provided
                        !   in a separate subprogram unit.
*
*   Initialize variables
*
      x_initial = 0.d0
      x_final   = 1.57d0 ! A close approximation to pi/2.
*
*   call the root-finding subroutine
*
      call Bisect( x_initial, x_final, Root, FofX )
*
      < print result, and end>

```

```

*-----
      Double Precision function FofX(x)
*
*   This is an example of a nonlinear function whose
*   root cannot be found analytically.
*
      Double Precision x
      FofX = cos(x) - x
      end
*-----

      Subroutine Bisect( Left, Right, middle, F )
*
      <prepare for looping:  set initial values, etc.>
      <TOP of the loop>
      ...
      <Body of the loop:  divide the interval in half,
                        determine which half contains the root,
                        redefine the limits of the bracket>
      ...
      IF (bracket still too big) go to the TOP of the loop
      ...
      < loop finished:
        put on finishing touches (if needed), and end>

```

The main program is almost complete, although we've yet to begin the root-finding subroutine itself! This is a general characteristic of the "top-down" programming we've described — first, write an outline for the overall design of the program, and then refine it successively until all the details have been worked out. In passing, we note that the *generic* cosine function is used in the function definition rather than the double precision function `dcos` — the generic functions, which include `abs`, `sin`, and `exp`, will always match the data type of the argument.

Now, let's concentrate on the root-finding subroutine. The beginning of the code might look something like the following:

```

*-----
      Subroutine Bisect( Left, Right, Middle, F )
*
*   Paul L. DeVries, Department of Physics, Miami University
*
*   Finding the root of f(x), known to be bracketed between
*   Left and Right, by the Bisection method. The root is
*   returned in the variable Middle.

```

```

*
*                               start date:    1/1/93
*
*  Type Declarations
*
*      DOUBLE PRECISION f, x
*      DOUBLE PRECISION Left, Right, Middle
*      DOUBLE PRECISION fLeft, fRight, fMiddle
*
*  Initialization of variables
*
*      fLeft   = f(Left)
*      fRight  = f(Right)
*      Middle  = (Left+Right)/2.d0
*      fMiddle = f(Middle)
*
*      ...

```

In addition to using reasonable names for variables, we've also tried to use selective capitalization to aid in the readability of the code. The single most important characteristic of your computer programs should be their clarity, and the readability of the code contributes significantly to that goal. The FORTRAN compiler does not distinguish between uppercase and lowercase, but we humans do!

How do we determine which subinterval contains the root? As in most puzzles, there are many ways we can find the answer. And the idea that occurs first is not necessarily the best. It's important to try various ideas, knowing that some will work and some won't. This particular puzzle has a well-known solution, however. If the root is in the left side, then `fLeft` and `fMiddle` will be of opposite sign and their product will be either zero or negative. So, if the expression `fLeft * fMiddle .le. 0` is true, the root is in the left side; if the expression is false, the root must be in the right side. *Voila!* Having determined which subinterval contains the root, we then redefine `Middle` to be `Right` if the root is in the left subinterval, or `Middle` to be `Left` if the root is in the right subinterval. This part of the code then looks like

```

*
*  Determine which half of the interval contains the root
*
*      IF (fLeft * fMiddle .le. 0 ) THEN
*
*          The root is in left subinterval:
*
*              Right = Middle

```

```

        fRight = fMiddle
    ELSE
    *
    *     The root is in right subinterval:
    *
        Left = Middle
        fLeft = fMiddle
    ENDIF
    *
    * The root is now bracketed between Left and Right!
    *
```

The IF-THEN-ELSE construct is ideally suited to the task at hand. The IF statement provides a condition to be met, in this case that the root lie in the left subinterval. If the condition is true, any statements following THEN are executed. If the condition is not true, any statements following ELSE are executed. The ENDIF completes the construct, and must be present.

Our pseudocode is quickly being replaced by actual code, but a critical part yet remains: how to terminate the process. Exactly what is the appropriate criterion for having found a root? Or to put it another way, what is the acceptable error in finding the root, and how is that expressed as an error condition?

Basically, there are two ways to quantify an error: in absolute terms, or in relative terms. The *absolute error* is simply the magnitude of the difference between the true value and the approximate value,

$$\text{Absolute error} = |\text{true value} - \text{approximate value}|. \quad (2.2)$$

Unfortunately, this measure of the error isn't as useful as you might think. Imagine two situations: in the first, the approximation is 1178.3 while the true value is 1178.4, and in the second situation the approximation is 0.15 while the true value is 0.25. In both cases the absolute error is 0.1, but clearly the approximation in the first case is better than in the second case. To gauge the *accuracy* of the approximation, we need to know more than merely the absolute error.

A better sense of the accuracy of an approximation is (usually) conveyed using a statement of *relative error*, comparing the difference between the approximation and the true value to the true value. That is,

$$\text{Relative Error} = \left| \frac{\text{True Value} - \text{Approximate Value}}{\text{True Value}} \right|. \quad (2.3)$$

The relative error in the first situation is $|\frac{0.1}{1178.4}| = 0.00008$, while in the second case it is $|\frac{0.1}{0.25}| = 0.4$. The higher accuracy of the first case is clearly associated with the much smaller relative error. Note, of course, that relative error is not defined if the true value is zero. And as a practical matter, the only quantity we can actually compute is an approximation to the relative error,

Approximate Relative Error

$$= \left| \frac{\text{Best Approximation} - \text{Previous Approximation}}{\text{Best Approximation}} \right|. \quad (2.4)$$

Thus, while there may be times when absolute accuracy is appropriate, most of the time we will want relative accuracy. For example, wanting to know x to within 1% is usually a more reasonable goal than simply wanting to know x to within 1, although this is not always true. (For example, in planning a trip to the Moon you might well want to know the distance to within 1 meter, not to within 1%!) Let's assume that in the present case our goal is to obtain results with relative error less than 5×10^{-8} . In this context, the "error" is simply our uncertainty in locating the root, which in the bisection method is just the width of the interval. (By the way, this accuracy in a trip to the Moon gives an absolute error of about 20 meters. Imagine being 20 meters above the surface, the descent rate of your lunar lander brought to zero, and out of gas. How hard would you hit the surface?) After declaring these additional variables, adding write statements, and cleaning up a few loose ends, the total code might look something like

```
*-----
      Subroutine Bisect( Left, Right, Middle, F )
*
* Paul L. DeVries, Department of Physics, Miami University
*
* Finding the root of the function "F" by BISECTION. The
* root is known(?) to be bracketed between LEFT and RIGHT.
*
*                               start date:    1/1/93
*
*
* Type Declarations
*
*      DOUBLE PRECISION    Left,  Right,  Middle
*      DOUBLE PRECISION f, fLeft, fRight, fMiddle
*      DOUBLE PRECISION TOL, Error
*
* Parameter declaration
```

```

*
*      PARAMETER( TOL = 5.d-03)
*
* Initialization of variables
*
*      FLeft   = f(Left)
*      FRight  = f(Right)
*
* Top of the Bisection loop
*
100  Middle = (Left+Right)/2.d0
      FMiddle = f(Middle)
*
* Determine which half of the interval contains the root
*
*      IF( fLeft * fMiddle .le. 0 ) THEN
*
*          The root is in left subinterval:
*
*              Right = Middle
*              fRight = fMiddle
*      ELSE
*
*          The root is in right subinterval:
*
*              Left = Middle
*              fLeft = fMiddle
*      ENDIF
*
* The root is now bracketed between (new) Left and Right !
*
* Check for the relative error condition:  If too big,
*      bisect again; if small enough, print result and end.
*
*      Error = ABS( (Right-Left)/Middle )
*      IF( Error .gt. TOL ) GOTO 100
*
*                                          ! Remove after
*      write(*,*) ' Root found at ',Middle ! routine has
*                                          ! been debugged.
*
*      end

```

We've introduced, and declared as DOUBLE PRECISION, the parameter TOL to describe the desired tolerance — you might recall that the parameter statement allows us to treat TOL as a named variable in FORTRAN statements but

prohibits us from accidentally changing its value. In the initial stages of writing and testing the program, TOL can be made relatively large and any lurking errors found quickly. Only after the code is known to be functioning properly should the tolerance be decreased to achieve the desired level of accuracy. Alternatively, we could pass the tolerance to the subroutine as a parameter. We have also coded a `write` statement in the subroutine. This is appropriate during the initial development phase, but should be removed — or “commented out” by inserting `*` in column 1 — after we’re satisfied that the subroutine is working properly. The purpose of this routine is to *find* the root — if writing it is desired, it should be done in a routine which calls `Bisect`, not in `Bisect` itself.

At the bottom of the loop, the relative error is computed. If the error is greater than the declared tolerance, another bisection step is performed; if not, the result is printed. To get to the top of the loop, a `GOTO` statement is used. Generally speaking, `GOTO`s should be avoided, as they encourage us to be sloppy in structuring programs. This can lead to programs that are virtually unreadable, making it easy for bugs to creep into them and more difficult for us to be rid of them. But in this instance — returning program control to the top of a loop — there is no suitable alternative. The program certainly looks like it’s ready to run. In fact, it’s foolproof!

Well, maybe not *foolproof*. It might happen that six months from now you need to find a root, and so you adopt this code. But what if you misjudge, or simply are incorrect, in your bracketing? As developed, the code *assumes* that there is a root between the initial `Left` and `Right`. In practice, finding such brackets might be rather difficult, and certainly calls for a different strategy than the one implemented here. Modify the code to include an explicit check that *verifies* that the root is bracketed. This verification should be performed after all the initialization has been accomplished but prior to the beginning of the main loop. In large, complex computer codes such *data validation* can become of preeminent importance to the overall success of the computation.

EXERCISE 2.1

Using the code we’ve developed, with your data validation, find the root of the equation $f(x) = \cos x - x = 0$ by the method of bisection. How many iterates are necessary to determine the root to 8 significant figures?

Well, bisection works, and it may be foolproof, but it certainly can be *slow*! It’s easy to determine how fast this method is. (Or slow, as the case may be.) Defining the error as the difference between the upper and lower bounds on

the root, at every iteration the error is halved. If ϵ_i is the error at the i -th step, at the next step we have $\epsilon_{i+1} = \epsilon_i/2$, and the rate of convergence is said to be *linear*. Given initial brackets, we could even determine how many iterations are necessary to obtain a specific level of accuracy.

It would seem that there would be a better way to find the root, at least for a nice, smoothly varying function such as ours. And indeed, there is such a method, due to Newton. But we need to use information about the function, and how it changes — that is, derivative information. That information is most succinctly provided in a Taylor series expansion of the function. Since this expansion is so central to much of what we are doing now, and will be doing in latter chapters, let's take a moment to review the essentials of the series.

Mr. Taylor's Series

In 1715 Brook Taylor, secretary of the Royal Society, published *Methodus Incrementorum Directa et Inversa* in which appears one of the most useful expressions in much of mathematics and certainly numerical analysis. Although previously known to the Scottish mathematician James Gregory, and probably to Jean Bernoulli as well, we know it today as the **Taylor series**. Since it will play a central role in many of our discussions, it's appropriate that we take a little time to discuss it in some detail. Let's assume that the function $f(x)$ has a continuous n th derivative in the interval $a \leq x \leq b$. We can then integrate this derivative to obtain

$$\int_a^{x_1} f^{[n]}(x_0) dx_0 = f^{[n-1]}(x_1) - f^{[n-1]}(a). \quad (2.5)$$

Integrating again, we have

$$\begin{aligned} \int_a^{x_2} \int_a^{x_1} f^{[n]}(x_0) dx_0 dx_1 &= \int_a^{x_2} \left(f^{[n-1]}(x_1) - f^{[n-1]}(a) \right) dx_1 \\ &= f^{[n-2]}(x_2) - f^{[n-2]}(a) - (x_2 - a)f^{[n-1]}(a). \end{aligned} \quad (2.6)$$

Continuing in this way we find, after n integrations,

$$\begin{aligned} \int_a^{x_n} \cdots \int_a^{x_1} f^{[n]}(x) dx_0 \cdots dx_{n-1} &= f(x_n) - f(a) - (x_n - a)f'(a) \\ &\quad - \frac{(x_n - a)^2}{2!} f''(a) - \frac{(x_n - a)^3}{3!} f'''(a) \cdots - \frac{(x_n - a)^{n-1}}{(n-1)!} f^{[n-1]}(a). \end{aligned} \quad (2.7)$$

To simplify the appearance of the expression, we now substitute x for x_n , and solve for $f(x)$ to obtain the *Taylor series*:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \frac{(x-a)^3}{3!}f'''(a) + \cdots + \frac{(x-a)^{n-1}}{(n-1)!}f^{[n-1]}(a) + R_n(x), \quad (2.8)$$

where

$$R_n(x) = \int_a^x \cdots \int_a^{x_1} f^{[n]}(x) dx_0 \cdots dx_{n-1}. \quad (2.9)$$

This remainder term is often written in a different way. Using the mean value theorem of integral calculus,

$$\int_a^x q(y) dy = (x-a)q(\xi), \quad \text{for } a \leq \xi \leq x, \quad (2.10)$$

and integrating $n-1$ more times, the remainder term can be written as

$$R_n(x) = \frac{(x-a)^n}{n!} f^{[n]}(\xi), \quad (2.11)$$

a form originally due to Lagrange. If the function is such that

$$\lim_{n \rightarrow \infty} R_n = 0, \quad (2.12)$$

then the finite series can be extended to an infinite number of terms and we arrive at the Taylor series expression for $f(x)$.

To illustrate, consider the Taylor series expansion of $\sin x$ about the point $x=0$:

$$\begin{aligned} f(x) &= \sin x, & f(0) &= 0, \\ f'(x) &= \cos x, & f'(0) &= 1, \\ f''(x) &= -\sin x, & f''(0) &= 0, \\ f''' &= -\cos x, & f'''(0) &= -1, \\ &\vdots & &\vdots \end{aligned} \quad (2.13)$$

with remainder

$$R_n(x) = \begin{cases} (-1)^{n/2} \frac{x^n}{n!} \sin \xi, & n \text{ even,} \\ (-1)^{(n-1)/2} \frac{x^n}{n!} \cos \xi, & n \text{ odd.} \end{cases} \quad (2.14)$$

Since the magnitudes of the sine and cosine are bounded by 1, the magnitude of the remainder satisfies the inequality

$$|R_n(x)| \leq \frac{x^n}{n!}. \quad (2.15)$$

For any given x , the factorial will eventually exceed the numerator and the remainder will tend toward zero. Thus we can expand $f(x) = \sin x$ as an infinite series,

$$\begin{aligned} f(x) &= f(0) + (x-0)f'(0) + \frac{(x-0)^2}{2!}f''(0) + \frac{(x-0)^3}{3!}f'''(0) + \cdots \\ &= x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots. \end{aligned} \quad (2.16)$$

This is, of course, simply the well-known approximation for the sine function.

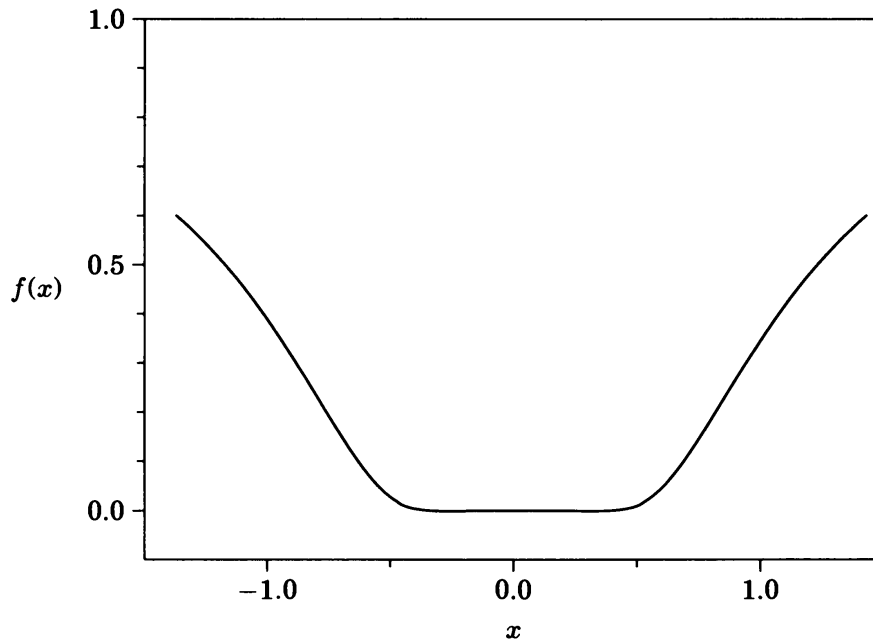


FIGURE 2.2 A decidedly unpleasant function!

Now, it might seem to be a small step to say that all functions that possess an infinite number of derivatives can be expressed as a Taylor series; but of course, falling off a cliff only takes a small step, too. Consider the function

$$f(x) = \begin{cases} e^{-1/x^2}, & \text{for } x \neq 0, \\ 0, & \text{for } x = 0, \end{cases}$$

which is plotted in Figure 2.2. This is *not* a “nasty” function — it is well-behaved, and goes smoothly to zero as $x \rightarrow 0$. In fact, it goes to zero so strongly that all its derivatives go to zero there as well. If we then try to use Taylor’s series about $a = 0$, we find that $f(x) = 0 + 0 + \cdots = 0$, *everywhere!*

The Newton–Raphson Method

To appreciate some of the power of Taylor’s series, we’ll use it to develop Newton’s method to find the zeros of a function. Assume that we have a good “guess,” so that $(x - a)$ is a small number. Then keeping just the first two terms of the Taylor series, we have

$$f(x) \approx f(a) + (x - a)f'(a). \quad (2.17)$$

We want to find that value of x for which $f(x) = 0$; setting $f(x)$ equal to zero and solving for x , we quickly find

$$x = a - \frac{f(a)}{f'(a)}. \quad (2.18)$$

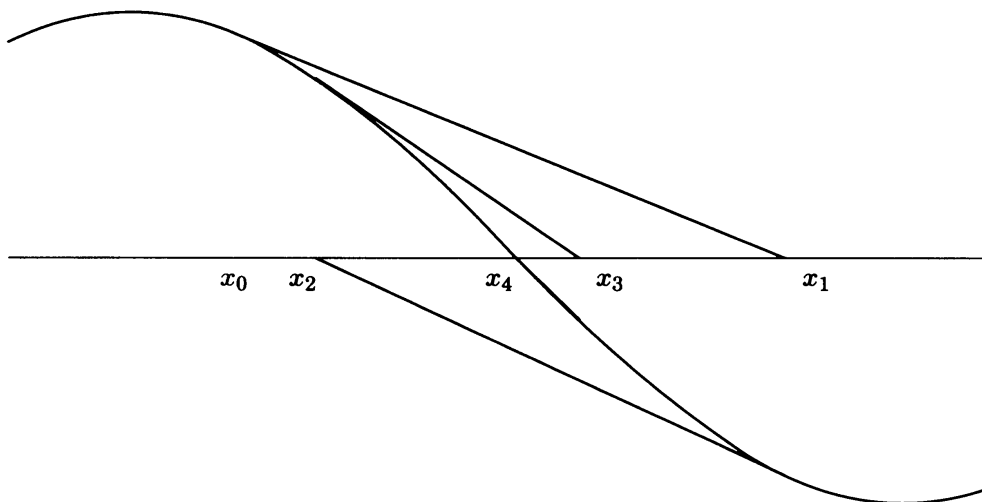


FIGURE 2.3 The Newton–Raphson in action. Beginning with x_0 , the successive iterates move closer to the zero of the function. The location of x_4 and the actual crossing are indistinguishable on this scale.

To see how this works, take a look at Figure 2.3. At $x = a$ the function and its derivative, which is tangent to the function, are known. Assuming that the function doesn't differ too much from a straight line, a good approximation to where the *function* crosses zero is where the *tangent line* crosses zero. This point, being the solution to a *linear* equation, is easily found — it's given by Equation (2.18)! Then this point can be taken as a new guess for the root, the function and its derivative evaluated, and so on. The idea of using one value to generate a better value is called *iteration*, and it is a very practical technique which we will use often. Changing the notation a little, we can calculate the $(i + 1)$ -th value x_{i+1} from the i -th value by the iterative expression

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \quad (2.19)$$

For the function $f(x) = \cos x - x$, we have that $f'(x) = -\sin x - 1$. We know that $\cos x = 1$ at $x = 0$, and that $\cos x = 0$ at $x = \pi/2$, so we might guess that $x = \cos x$ somewhere around $x_0 = \pi/4$. We then calculate the zero to be at

$$x_1 = \frac{\pi}{4} - \frac{\cos \pi/4 - \pi/4}{-\sin \pi/4 - 1} = 0.739536134. \quad (2.20)$$

This is a pretty good result, much closer to the correct answer of 0.739085133 than was the initial guess of $\pi/4 = 0.785398163$. And as noted, this result can be used as a new guess to calculate another approximation to the location of the zero. Thus

$$x_2 = 0.739536134 - \frac{\cos(0.739536134) - 0.739536134}{-\sin(0.739536134) - 1} = 0.739085178, \quad (2.21)$$

a result accurate to 7 significant digits.

Beginning with an initial guess x_0 , the expression is iterated to generate x_1, x_2, \dots , until the result is deemed sufficiently accurate. Typically we want a result that is accurate to about eight significant digits, e.g., a relative error of 5×10^{-8} . That means that after each evaluation of x_{i+1} it should be compared to x_i ; if the desired accuracy has been obtained, we should quit. On the other hand, if the accuracy has not been obtained, we should iterate again. Since we don't know how many iterations will be needed, a DO loop is not appropriate for this task. Rather, we need to implement a loop with a GOTO statement, being sure to include an exit out of the loop after the error condition has been met.

```
Program ROOTS
Double Precision x, FofX, DERofF
```

```

      External FofX, DERofF
      x = 0.8d0
      call Newton ( x, FofX, DERofF )
      write(*,*) ' Root found at x =',x
      end
*-----
      Double Precision Function FofX(x)
*
*   This is an example of a nonlinear function whose
*   root cannot be found analytically.
*
      Double Precision x
      FofX = cos(x) - x
      end
*-----
      Double Precision Function DERofF(x)
*
*   This function is the derivative of "F of X."
*
      Double Precision x
      DERofF = -sin(x) - 1.d0
      end
*-----
      Subroutine Newton ( x, F, Fprime )
*
*   Paul L. DeVries, Department of Physics, Miami University
*
*   Preliminary code for root finding with Newton-Raphson
*
*               start date:    1/1/93
*
*   Type declarations
*
      DOUBLE PRECISION x, F, Fprime, delta, error, TOL
*
*   Parameter declarations
*
      PARAMETER(  TOL = 5.d-03)
*
*   Top of the loop
*
100  delta = -f(x)/fprime(x)
      x = x + delta
*

```

```

* Check for the relative error condition: If too big,
* loop again; if small enough, end.
*
      Error = ABS( delta / x )
      IF ( Error .gt. TOL) GOTO 100
      end

```

EXERCISE 2.2

Verify that this code is functioning properly by finding (again) where $x = \cos x$. Compare the effort required to find the root with the Newton–Raphson and the bisection methods.

As we noted earlier, finding the roots of equations often occurs in a larger context. For example, in Chapter 4 we will find that the zeros of Legendre functions play a special role in certain integration schemes. So, let's consider the Legendre polynomial

$$P_8(x) = \frac{6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35}{128}, \quad -1 \leq x \leq 1, \quad (2.22)$$

and try to find its roots. Where to start? What would be a good initial guess? Since only the first derivative term was retained in developing the Newton–Raphson method, we suspect that we need to be close to a root before using it. For $|x| < 1$, we have $x^8 < x^6 < x^4 < x^2$. Let's (temporarily) ignore all the terms in the polynomial except the last two, and set this truncated function equal to zero. We thus have

$$P_8(x_0) \approx \frac{-1260x_0^2 + 35}{128} = 0, \quad (2.23)$$

and hence

$$x_0 = \pm \sqrt{\frac{35}{1260}} = \pm \sqrt{\frac{1}{36}} = \pm \frac{1}{6}. \quad (2.24)$$

Thus 0.167 should be an excellent guess to begin the iteration for the smallest non-negative root.

EXERCISE 2.3

Use the Newton–Raphson method to find the smallest non-negative root of $P_8(x)$.

Root-finding in general, and the Newton–Raphson method in particular, arise in various rather unexpected places. For example, how could we

find the two-thirds power of 13, with only a 4-function hand calculator? That is, we want to find x such that

$$x = 13^{2/3}. \quad (2.25)$$

Cubing both sides, we have

$$x^3 = 13^2 = 169 \quad (2.26)$$

or

$$x^3 - 169 = 0. \quad (2.27)$$

Interesting.

EXERCISE 2.4

Use the Newton–Raphson method to solve Equation (2.27) for the two-thirds root of 13, to 8 significant figures.

Fools Rush In ...

Well, our Newton–Raphson method seems to be working so well, let’s try it to find a root of the equation $f(x) = x^2 + 1 = 0$. No problem. Starting the iteration with $x_0 = 1/\sqrt{3}$, we find the first few iterates to be

$$\begin{aligned} x_0 &= .577350269 \\ x_1 &= -.577350269 \\ x_2 &= .577350269 \\ x_3 &= -.577350269 \\ x_4 &= .577350269 \\ x_5 &= -.577350269 \\ &\vdots \end{aligned}$$

Oops.

Clearly, there’s a problem here. Not only are the iterates not converging, they’re just flopping back and forth between $\pm 1/\sqrt{3}$. This example forces us to realize that we haven’t yet thought enough about how to approach computing. Perhaps the first thing we need to realize is that the world is not always nice; sometimes things just don’t go our way. While we always try to do things right, errors will always be made, and we can’t expect the computer to protect us from ourselves. In particular, the computer can’t do something

we haven't instructed it to do — it's not smarter than we are. So, just because the computer is there and is eager to help us if we only type in the code, we must first decide if the problem is suitable for computation. Perhaps it needs to be transformed into an equivalent problem, stated differently; perhaps another algorithm should be used; or perhaps we should have realized that the problem at hand has no real roots.

Computing is not a substitute for thinking.

As we write programs to solve problems of interest, we'll try to anticipate various situations. This will probably require extra programming for special cases. In the problem above, we might have tried to determine if a real root existed before we tried to find it. Certainly we can't expect the program to know what we don't. What we need is a strategy toward computing, a *game plan*, so to speak. The Newton–Raphson method is a robust algorithm that often works very well — it's like a high-risk *offense*, capable of aggressively finding a root. What we've missed is what every sports fan knows: while offense wins games, *defense* wins championships. We should strive to be intelligent in our computing, anticipating various possibilities, and trying not to make errors. But we should be prepared to fall short of this goal — it's simply not possible to foresee all eventualities. Reasonable people know that mistakes will occasionally be made, and take appropriate steps *before* they occur. If we can't instruct the computer how always to act correctly, let's at least instruct it how not to act incorrectly! A major key to successful programming is to

Compute defensively!

But how do we instruct the computer not to act incorrectly? In this case, it's pretty easy. When the iteration worked, it was very fast. But when it failed, it failed miserably. Thus we can require that the algorithm either finds a root in just a few steps, or quits. Of course, if it quits it should inform us that it is quitting because it hasn't found a root. (There's nothing quite as frustrating as a computer program that stops with no explanation. Or one that prints ERROR, without any indication of what kind of error or where it occurred.) The maximum number of iterates could be passed to the code fragment, but it is easier to just set a maximum that seems much greater than you would ever need, say 30 iterations. The idea is to prevent infinite loops and to provide a graceful exit for the program.

EXERCISE 2.5

Modify the displayed code to keep track of the number of iterations, and to exit the loop and end after writing an appropriate message if a root is not found in 30 iterations. Test your modifications on the example above, $f(x) = x^2 + 1$, with an initial x of 0.5. You might want to print out the iterates, just to see how the search proceeds.

Rates of Convergence

We previously saw that the bisection method was *linear* in its convergence. Our experience with the Newton–Raphson scheme is that if it converges, it converges very rapidly. To quantify this statement, let’s return to the Taylor series

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + \frac{(x - x_i)^2}{2!}f''(x_i) + \cdots \quad (2.28)$$

We’ll assume that we’re in a region where the function is well behaved and the first derivative is not small. Keeping the first two terms and setting $f(x) = 0$, we’re led to the iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \quad (2.29)$$

just as before. Recall that both x_i and x_{i+1} are approximations to the location of the root — if x is where the root *actually* is, then $\epsilon_i = x - x_i$ is a measure of the error in x_i . Subtracting x from both sides of Equation (2.29), we find

$$\epsilon_{i+1} = \epsilon_i + \frac{f(x_i)}{f'(x_i)}, \quad (2.30)$$

an expression relating the error at one iteration to the next. Let’s return to the Taylor series, this time keeping the first three terms in the series. Setting $f(x) = 0$ and solving for $f(x_i)$, we find

$$f(x_i) = -\epsilon_i f'(x_i) - \frac{\epsilon_i^2}{2} f''(x_i). \quad (2.31)$$

Substituting this expression into the previous one, we find that

$$\begin{aligned}\epsilon_{i+1} &= \epsilon_i + \frac{-\epsilon_i f'(x_i) - \frac{\epsilon_i^2}{2} f''(x_i)}{f'(x_i)} \\ &= -\frac{\epsilon_i^2 f''(x_i)}{2f'(x_i)}.\end{aligned}\tag{2.32}$$

If f''/f' is approximately constant near the root, then at each step the error is proportional to the *square* of the error at the previous step — if the error is initially small, it gets smaller very quickly. Such convergence is termed *quadratic*. In the Newton–Raphson iteration, each iteration essentially doubles the number of significant figures that are accurate. (It is possible, however, for the method not to converge: this might happen if the initial error is too large, for example. It’s also possible that f' is near zero in the vicinity of the root, which happens if there are two roots of $f(x)$ close to one another. Could you modify the method presented here to treat that situation?)

It would be desirable to combine the slow-but-sure quality of the bisection method with the fast-but-iffy characteristic of Newton’s method to come up with a guaranteed winner. With a little thought, we can do just that. Let’s start with the fact (verified?) that the root is bounded between $x = a$ and $x = c$, and that the best current guess for the root is $x = b$. (Note that initially we may choose b to be either a or c .) We’ll include the iteration counter, so that the code doesn’t loop forever. But the critical step is to decide when to take a bisection step, and when to take a Newton–Raphson step. Clearly, we want to take a Newton–Raphson wherever possible. Since we *know* that the root is bracketed, it’s clearly in our interest to only take steps that lie within those limits. Thus the crucial step is to determine if the *next Newton–Raphson guess* will be outside the bounding interval or not. If the next guess is within the bounds, then

$$a \leq b - \frac{f(b)}{f'(b)} \leq c.\tag{2.33}$$

Subtracting b from all terms, multiplying by $-f'(b)$, and subtracting $f(b)$, this inequality can be rewritten as

$$(b - a)f'(b) - f(b) \geq 0 \geq (b - c)f'(b) - f(b).\tag{2.34}$$

This inequality will be satisfied *if* the next iterate lies between a and c . An easy way of determining if this is so is to compare the product of the first term and the last term to zero. If the product is less than or equal to zero, then the next Newton–Raphson guess will fall within the known limits, and so a

Newton–Raphson step should certainly be taken; if this condition is not met, a bisection step should be taken instead. Now, to attempt a judicious merger of the bisection and Newton–Raphson methods using this logic to decide which method to utilize ...

```

      Program MoreROOTS
      Double Precision x_initial, x_final, x_root, FofX,
+          DERofF
      External FofX, DERofF
      x_initial = ...
      x_final   = ...
      call Hybrid(x_initial, x_final, x_root, FofX, DERofF)
      write(*,*) ' root at x =',x_root
      end

*-----
      Double Precision Function FofX(x)
      Double Precision x

*
* < Define the function >
*
      end

*-----
      Double Precision Function DERofF(x)
      Double Precision x

*
* < Define the derivative of the function. >
*
      end

*-----
      Subroutine Hybrid ( Left, Right, Best, F, Fprime )
*
* Paul L. DeVries, Department of Physics, Miami University
*
* A hybrid BISECTION/NEWTON-RAPHSON method for
* finding the root of a function F, with derivative
* Fprime. The root is known to be between Left and
* Right, and the result is returned in 'best'. If the
* next NR guess is within the known bounds, the step
* is accepted; otherwise, a bisection step is taken.
*
*
* start date: 1/1/93
*
* The root is initially bracketed between Left and Right:
*
```

```

*           x :           Left           Best           Right
*           f(x):          fLeft          fBest          fRight
*           f'(x):          ---           DerfBest          ---
*
* Type Declarations
*
*           Double Precision f,fprime,Left,fLeft,Right,fRight,
+           Best,fBest,DerfBest,delta,TOL
*           integer count
*
* Initialize parameters
*
*           Parameter (TOL = 5.d-3)
*
* Initialization of variables
*
*           fLeft  = f(Left)
*           fRight = f(Right)
*
* Verify that root is bracketed:
*
*           IF(fLeft * fRight .gt. 0)STOP 'root NOT bracketed'
*
* Just to get started, let BEST = ...
*
*           IF( abs(fLeft).le.abs(fRight) ) THEN
*               Best = Left
*               fBest = fLeft
*           ELSE
*               Best = Right
*               fBest = fRight
*           ENDIF
*           DerfBest = fprime(Best)
*
* COUNT is the number of times through the loop
*
*           count = 0
100    count = count + 1
*
* Determine Newton-Raphson or Bisection step:
*
*           IF(      ( DerfBest * (Best-Left ) - fBest) *
+           ( DerfBest * (Best-Right) - fBest) .le. 0)
+ THEN

```

```

*
*   O.K. to take a Newton-Raphson step
*
*       delta = -fBest/DerfBest
*       Best  = Best + delta
*   ELSE
*
*   take a bisection step instead
*
*       delta = (Right-Left)/2.d0
*       Best  = (Left+Right)/2.d0
*   ENDIF
*
*   Compare the relative error to the TOLerance
*
*       IF( abs(delta/Best) .le. TOL ) THEN
*
*   Error is BELOW tolerance, the root has been found!
*
*       write(*,*)'root found at ',Best
*   ELSE
*
*   The relative error is too big, prepare to loop again ...
*
*       fBest = f(Best)
*       DerfBest = fprime(Best)
*
*   Adjust brackets
*
*       IF( fLeft * fBest .le. 0 ) THEN
*   The root is in left subinterval:
*
*       Right = Best
*       fRight = fBest
*
*   ELSE
*   The root is in right subinterval:
*
*       Left = Best
*       fLeft = fBest
*   ENDIF
*
*   Test for iteration count:
*

```

```

        IF( COUNT .lt. 30 ) goto 100
*
*   Can only get to this point if the ERROR is TOO BIG
*   and if COUNT is greater than 30. Time to QUIT !!!
*
        STOP 'Root not converged after 30 iterations.'
ENDIF
end

```

This looks like it just might work . . .

■ EXERCISE 2.6

Use the hybrid method discussed above to find the root of the equation

$$x^2 - 2x - 2 = 0,$$

given that there is a root between 0 and 3. You might find it interesting to add write statements indicating whether a Newton–Raphson or bisection step is being taken. Such statements can be a great help in writing and debugging programs, and can then be commented out when no longer needed. (And easily reinstated if a problem in the code later develops!)

We should note that in our implementation, STOP statements were encountered if *i*) the root was not initially bracketed or if *ii*) the result hadn't converged after 30 iterations. These statements will cause the execution of the program to terminate. In a large, long-running program it might be preferable for the program to *continue* to execute, but perhaps to take some other action if these conditions have been met. To achieve that, we can introduce an *error flag*. Typically an integer, the flag would be passed to Hybrid in the argument list, initialized to zero, and the STOP statements replaced by statements assigning a value to the flag. For example, if the root were not bracketed, the flag could be set to 1; if the result didn't converge, the flag could be set to 2. If either of these conditions were met, the value of the flag would change but the program would continue to execute. Control statements would then be added to ROOTS, after the call to Hybrid, to test the value of the flag: the flag being zero would indicate that the subroutine had executed as expected, but a nonzero flag would indicate an error. Appropriate action, redefining the initial interval, for example, could then be taken, depending upon the error that had been encountered.

Exhaustive Searching

We now have a good method for finding a root, if you first know that the root is bounded. But how do you find those bounds? Unfortunately, the answer is that there is *no good way* of finding them. The reason, of course, is that finding the bounds is a *global* problem — the root might be found *anywhere* — but finding the root, after the bounds are known, is a *local* problem. Almost by definition, local problems are always easier to solve than global ones.

So, what to do? One possibility is to graph the curve, and let your eye find the bounds. This is highly recommended, and for simple functions can be done with pencil and paper; for more complicated ones, the computer can be used to graph the function for you. But we can also investigate the function analytically. When we were finding an initial guess for the root of the Legendre function $P_8(x)$, we knew that all the roots were less than 1. As a result, $x^8 < x^6 < x^4 < x^2$, so we ignored the leading terms in the polynomial. Keeping only the two largest terms, we could exactly solve for the root of the truncated function. In a similar fashion, we can obtain a good guess for the largest root of a polynomial, if it's greater than 1. To illustrate, consider the quadratic equation

$$f(x) = x^2 - 11x + 10. \quad (2.35)$$

This can of course be solved exactly, and roots found at $x = 1$ and 10 . But keeping just the leading two terms, we have the truncated function

$$\bar{f}(x) = x^2 - 11x, \quad (2.36)$$

which has a root at

$$x = 11, \quad (2.37)$$

a very reasonable approximation to the root of the original function, obtained without taking a square root. Of course, the value of the approximation is more impressive if the problem isn't quite so obvious. Consider, for example, the function

$$f(x) = x^3 - 7x^2 - 10x + 16. \quad (2.38)$$

Our approximation to the largest root is 7 — but what is the exact value?

While there are shortcuts for polynomials — many more than we've mentioned — there are no similar tricks for general functions. The only method that is generally applicable is brute force, exhaustive searching. The most common strategy is simply to step along, evaluating the function at each step and determining if the function has changed sign. If it has, then a root is located within that step; if not, the search is continued. The problem with this procedure, of course, is that the step might be so large that the function has changed sign *twice*, e.g., there are two roots in the interval, in which case this

method will not detect their presence. (If derivatives are easily obtained, that information can be incorporated in the search by determining if the derivative has changed sign. If it has, that *suggests* that there might be two roots in the interval — at least, there is a minimum.)

Of course, determining an appropriate-sized step is as much educated guesswork as anything else. Use any and all information you have about the function, and be conservative. For example, we know that there are 4 roots of $P_8(x)$ between 0 and 1. If they were equally distributed, they might be 0.333 apart. Any search should certainly use a step at least half this, or 0.167, if not smaller.

EXERCISE 2.7

Find all the non-negative roots of $P_8(x)$. Use an exhaustive search to isolate the roots, and then use the hybrid algorithm you've developed to locate the roots themselves to a relative accuracy of 5×10^{-8} .

Look, Ma, No Derivatives!

With the safeguards we've added, the hybrid Bisection/Newton–Raphson algorithm you've developed is a good one. Unfortunately, it often happens that the required derivatives are not directly available to us. (Or perhaps the function is available but is so complicated that obtaining its derivative is difficult; or having the derivative, getting it coded without errors is unlikely.) For those situations, we need a method that requires only evaluations of the function, and not its derivatives, such as the bisection method.

Well, bisection works, but it certainly can be slow! The reason, of course, is that we have *not* been very resourceful in using the information available to us. After all, we know (assume?) that the root is bounded, $a \leq \bar{x} \leq c$, and we know the value of the function at the limits of the interval. Let's use this information as best we can, and approximate $f(x)$ by the straight line passing through the known values of the function at the endpoints of the interval, $[a, f(a)]$ and $[c, f(c)]$. This leads to the *method of false position*. You can easily verify that the line is given by the equation

$$p(x) = \frac{x - c}{a - c}f(a) + \frac{x - a}{c - a}f(c) \quad (2.39)$$

and can be thought to be an *approximation* to the function $f(x)$. Being a

simpler equation, its root can be easily determined: setting $p(\bar{x}) = 0$, we find

$$\bar{x} = \frac{af(c) - cf(a)}{f(c) - f(a)}. \quad (2.40)$$

This approximation is depicted graphically in Figure 2.4. Since $f(a)$ and $f(c)$ are opposite in sign, there's no worry that the denominator might vanish. It should be fairly obvious that in many common circumstances this is a considerable improvement over the bisection method. (An interesting analogy can be found in finding someone's phone number in the telephone book. To find Arnold Aaron's telephone number, the "bisection" method would first look under *M*, halfway in the book. It would then divide the interval in half, and look under *G*, about a quarter of the way through the book, and so on. In contrast, the method of false position would expect to find Arnold's name early in the phone book, and would start looking only a few pages into it.)

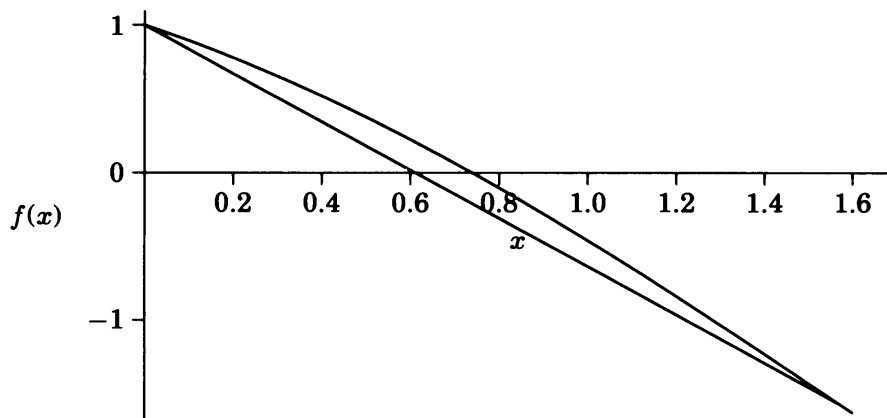


FIGURE 2.4 The function $f(x) = \cos x - x$ and its linear approximation, $p(x)$.

After the guess is made, the iteration proceeds by determining which subinterval contains the root and adjusting the bounds a and c appropriately. Since only one of the endpoints is changed, it is entirely possible, even likely, that one of these points will be fixed. (Is this statement obviously true? If not, look at the figure once again.) For example, the method might converge to the root from above, so that each iteration leaves c closer and closer to the root, *but never changes a !* What are important are the successive guesses, and the difference in them (à la Newton–Raphson) rather than the difference between the bounds. Thus, you must keep track of x_i and x_{i+1} as well as a and c .

Beginning with your existing bisection code, only a few changes are

necessary to transform it into a false position code. The first, of course, is to modify the expression for the root: instead of the variable `Middle`, we'll use `NewGuess`, and define it according to Equation (2.40). Also, we need to keep tabs on the successive approximations: we'll use `OldGuess` as the previous approximation. An outline of the code might look like

```

      Subroutine False ( Left, Right, NewGuess, F )
*
* < prepare for looping:  set initial values, etc. To
*   get started, must assign some value to OldGuess. >
*
      OldGuess = right
* < TOP of loop >
      NewGuess = ( Left * fRight - Right * fLeft)
+               / (fRight - fLeft)
      fGuess = f(NewGuess)
      Error = ABS ( (NewGuess - OldGuess)/NewGuess )
      IF( Error .gt. TOL )THEN
*         <determine which subinterval contains
*           the root, and redefine Left and Right
*           accordingly>
      OldGuess = NewGuess
*         < go to TOP of loop>
      ELSE
*         < NewGuess is a good approximation to
*           the root. >
      ENDIF
      end

```

EXERCISE 2.8

Modify your old bisection code to use the method of false position. Use the new code to find the root of $f(x) = x - \cos x = 0$, and compare the effort required, i.e., the number of iterations, to that of the bisection method.

The method of false position appears to be an obvious improvement over the bisection method, but there are some interesting situations in which that's not so. Consider the example in Figure 2.5. Although the root is bracketed, the bounds aren't close enough to the root to justify a linear approximation for the function. In this case, the method of false position might actually require *more* function evaluations than bisection.

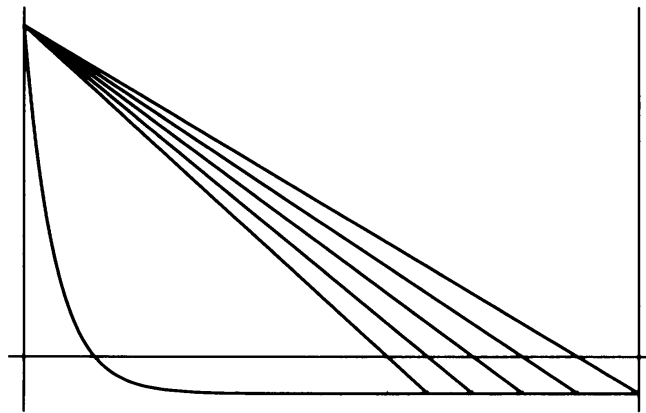


FIGURE 2.5 An example illustrating that the convergence of the method of false position can be lethargic.

We can rewrite Equation (2.40) as

$$\bar{x} = a - f(a) \frac{c - a}{f(c) - f(a)}, \quad (2.41)$$

which looks suspiciously like Newton–Raphson with the derivative approximated by

$$f'(a) \approx \frac{f(c) - f(a)}{c - a}. \quad (2.42)$$

Recall, however, that one of the endpoints will be fixed as we approach the root. That is, c might be fixed as a approaches the root. Then $c - a$ approaches a constant, but not zero. Although Equation (2.42) looks something like a derivative, this expression *does not* approach the actual derivative in the limit.

However, the successive iterative approximations do approach one another, so that *they* can be used to approximate the derivative. Using the previous two iterations for the root to approximate the derivative appearing in Newton’s method gives us the *Secant method*. That is, if x_i and x_{i-1} are the previous two approximations to the root, then the next approximation, x_{i+1} , is given as

$$x_{i+1} = x_i - f(x_i) \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}. \quad (2.43)$$

As with the Newton–Raphson method, the Secant method works very well when it works, but it’s not guaranteed to converge. Clearly, a hybrid combination of the Bisection and Secant methods will provide the superior method for finding roots when explicit derivative information is not available.

EXERCISE 2.9

Replace Newton–Raphson by Secant in the Hybrid code, so that only function evaluations — no derivatives — are used to find the root of a function. Use this method to find a root of $f(x) = \sin x - x/2 = 0$ between $\pi/2$ and π .

Accelerating the Rate of Convergence

For the exercises you’ve seen so far, and for many similar problems, the speed with which the computer finds a root is not really an issue: the time required to find a solution has all been in the development, writing, and debugging of the computer program, not in the actual calculation. But effort spent in developing good, reliable methods is spent only once — you’re now learning how to find roots, and that effort shouldn’t be duplicated in the future. In fact, you should regard your current effort as an investment for the future, to be paid off when you desperately need a solution to a particular problem and realize you already have all the tools needed to obtain it.

In the real world we usually find that the problems that interest us become more complicated, and the functions get harder to evaluate. As a result, the time required to calculate a solution gets longer. Thus the efficiency of a root finder should be discussed in terms of the number of function evaluations required, not in the complexity of the root-finding algorithm. For example, perhaps the “function” whose zero you’re trying to find is actually a multidimensional integral that requires an hour of supercomputer time to evaluate. Not having a supercomputer at your disposal, that’s fifty hours on your local mainframe, or ten days on your microcomputer. Very quickly, you realize that life will pass you by if you can’t find that root *Real Soon Now!* And a few extra multiplications to find that root, at microseconds each, don’t add much to the total time involved.

Thus, we want a root, and we want it with as few function, and/or derivative, evaluations as possible. The key is to realize that we’ve been all too eager to discard expensive, hard-to-obtain information — let’s see if we can’t use some of those function evaluations that we’ve been throwing away. For concreteness, let’s look at the Secant method a little more closely.

The harder information is to obtain, the more reluctant we should be to discard it.

In the Hybrid Bisection/Secant method we've developed, a linear approximation is used to obtain the next approximation to the root, and then the endpoints of the interval are adjusted to keep the root bounded. By using three points, which we have, a quadratic could be fitted to the function. We would then have a quadratic approximation for the root rather than a linear one, and *with no additional function evaluations*. Sounds good.

Consider the points x_0 , x_1 , and x_2 , and the function evaluated at these points. We can think of these as three successive approximations to the root of the function $f(x)$. The quadratic

$$p(x) = a(x - x_2)^2 + b(x - x_2) + c \quad (2.44)$$

will pass through these points if

$$\begin{aligned} c &= f(x_2), \\ b &= \frac{(x_0 - x_2)^2[f(x_1) - f(x_2)] - (x_1 - x_2)^2[f(x_0) - f(x_2)]}{(x_0 - x_1)(x_0 - x_2)(x_1 - x_2)}, \\ a &= \frac{(x_1 - x_2)[f(x_0) - f(x_2)] - (x_0 - x_2)[f(x_1) - f(x_2)]}{(x_0 - x_1)(x_0 - x_2)(x_1 - x_2)}. \end{aligned} \quad (2.45)$$

The next approximation to the root, x_3 , is then found by setting $p(x_3) = 0$ and solving the *quadratic* equation to find

$$x_3 - x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (2.46)$$

We expect (or, at least, we hope) that we are converging to the root, so that $x_3 - x_2$ is small in magnitude. But according to Equation (2.46), this will only happen if $-b$ and $\pm\sqrt{b^2 - 4ac}$ are nearly equal in magnitude and opposite in sign. That is, the small magnitude will be achieved by taking the difference between two nearly equal numbers, not a very sound practice. Rather than evaluating the root by Equation (2.46), let's develop an alternate form that has better numerical characteristics.

Let's assume for the moment that $b \geq 0$ — then the root will be associated with the plus sign in the expression, and we can rewrite Equation

(2.46) as

$$\begin{aligned} x_3 - x_2 &= \left(\frac{-b + \sqrt{b^2 - 4ac}}{2a} \right) \left(\frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \right) \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b - \sqrt{b^2 - 4ac})} = -\frac{2c}{b + \sqrt{b^2 - 4ac}}, \quad b \geq 0. \end{aligned} \quad (2.47)$$

For $b \leq 0$, the same reasoning leads to

$$\begin{aligned} x_3 - x_2 &= \left(\frac{-b - \sqrt{b^2 - 4ac}}{2a} \right) \left(\frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} \right) \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b + \sqrt{b^2 - 4ac})} = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \quad b \leq 0. \end{aligned} \quad (2.48)$$

It should be fairly clear that the Hybrid Bisection/Secant method can easily be modified to incorporate this quadratic approximation to find the root. The quadratic approximation itself is due to Müller; coupling it with the Bisection method is due to Brent. The result is a robust, virtually failsafe method of finding roots using only function evaluations.

EXERCISE 2.10

Modify your code to include this accelerated method. Verify that it's functioning correctly by finding x such that

$$\cos x = x \sin x. \quad (2.49)$$

Before leaving this section, let's see if we can't extract a little more information from our expression for the root. Differentiating (2.44), we find that

$$p'(x) = 2a(x - x_2) + b, \quad (2.50)$$

so that

$$p'(x_2) = b. \quad (2.51)$$

Near the root, we'll assume that the derivative is nonzero, while the function itself is small. That is, $b = p'(x_2) \neq 0$ while $c = p(x_2) \approx 0$. We can then write

$$x_3 - x_2 = -\frac{2c}{b + \sqrt{b^2 - 4ac}} = -\frac{2c}{b + b\sqrt{1 - 4ac/b^2}}, \quad b \geq 0, \quad (2.52)$$

where $4ac/b^2$ is a small term. Making small argument expansions, we find

$$\begin{aligned}
 x_3 - x_2 &\approx -\frac{2c}{b + b(1 - 2ac/b^2 + \dots)} \approx -\frac{2c}{2b} \frac{1}{1 - ac/b^2} \\
 &\approx -\frac{c}{b} \left(1 + \frac{ac}{b^2} + \dots\right) \approx -\frac{c}{b} - \frac{ac^2}{b^3} \\
 &= -\frac{f(x_2)}{f'(x_2)} - \frac{f''(x_2)f^2(x_2)}{2[f'(x_2)]^3}.
 \end{aligned} \tag{2.53}$$

The first term we recognize as just the Newton–Raphson expression — the second is a *correction term*, partially accounting for the nonlinearity of the actual function. (That is, a quadratic contribution.) This expression has an interesting geometrical interpretation. Rather than simply putting a straight line through $[x_2, f(x_2)]$, the line is drawn through a point that is moved up or down to compensate for the shape of the function, as seen in Figure 2.6. The line that’s drawn is still linear, but the curvature of the function has been taken into account.

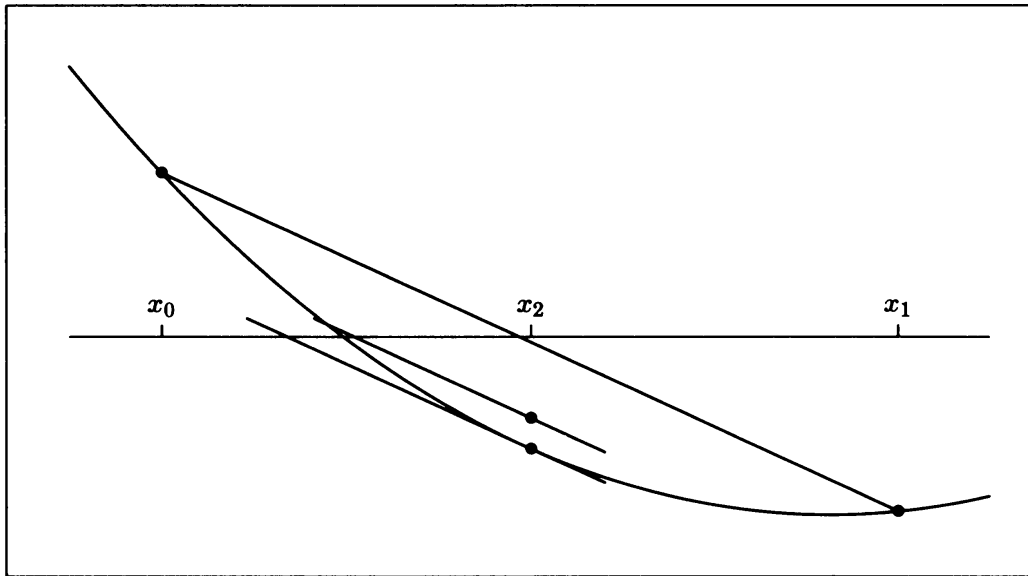


FIGURE 2.6 Accelerating the rate of convergence.

A Little Quantum Mechanics Problem

Our interest is not in numerical methods *per se*, but in investigating physical processes and solving physical problems. The only reason we’ve been looking at the roots of equations is because there are problems of interest to us that

present themselves in this way. One such problem arises in quantum mechanics and is routinely used for illustration in elementary textbooks: finding the eigenvalues of the finite square well.

Newton's equations of motion play a fundamental role in classical mechanics; in quantum mechanics, that role is played by the Schrödinger equation,

$$-\frac{\hbar^2}{2m} \frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x). \quad (2.54)$$

This is the equation that determines “all there is to know” about the one-dimensional problem of a particle of mass m moving in the potential $V(x)$. In this equation, \hbar is a fundamental constant due to Planck, E is the total energy, and ψ is the *wavefunction* of the system. ψ is the unknown quantity that you're solving for, the “solution” to the Schrödinger equation. Once ψ is determined, all the observable quantities pertaining to the system can be calculated. In particular, $\psi^*\psi dx$ is the probability of finding the particle between x and $x + dx$.

Now comes the hard part: it can happen that E is unknown as well! It would seem that there are too many unknowns in the Schrödinger equation for us to solve the problem, and that would be true except for certain physical constraints that we impose. Let's consider a specific example, the infinite square well.

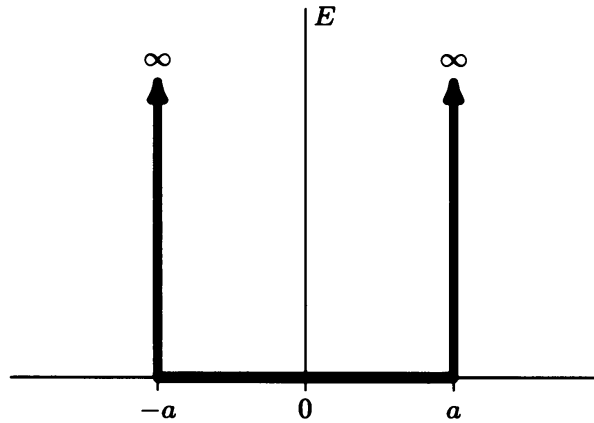


FIGURE 2.7 The infinite square well.

As illustrated in Figure 2.7, the potential for the infinite square well is zero between $-a$ and a , but infinite outside that region. The Schrödinger equation can be solved in the interior of the well with the general result that

$$\psi(x) = A \sin kx + B \cos kx, \quad (2.55)$$

where

$$k = \sqrt{\frac{2mE}{\hbar^2}}. \quad (2.56)$$

Outside the interval $-a \leq x \leq a$, the potential is infinite and so we shouldn't expect to find the particle there. If the particle can't be found there, then the wavefunction is zero in that region. Since we also expect that the probability of finding a particle is a continuous function, we require that the wavefunction vanishes at $\pm a$. That is, the *physics* of the problem *requires* that the wavefunction vanish at these points. We have thus established *boundary conditions* that must be satisfied by any mathematical solution in order to be physically correct.

Now we need to *impose* these boundary conditions upon the general solution. At $x = -a$, we *require* that

$$-A \sin ka + B \cos ka = 0, \quad (2.57)$$

while at $x = +a$, we require that

$$A \sin ka + B \cos ka = 0. \quad (2.58)$$

We now add and subtract these two equations, to obtain

$$B \cos ka = 0 \quad (2.59)$$

and

$$A \sin ka = 0. \quad (2.60)$$

Consider the second equation, $A \sin ka = 0$. This can be accomplished in one of two ways: either A or $\sin ka$ is identically zero. Let's take $A = 0$. Then, if the wavefunction is to be at all interesting, $B \neq 0$. But $B \cos ka = 0$, and if $B \neq 0$, then we must have $\cos ka = 0$. The only way the cosine can vanish is if the argument is equal to $\pi/2, 3\pi/2, \dots$. That is, we've found that the boundary condition can be met if

$$ka = \sqrt{\frac{2mE}{\hbar^2}}a = (n + 1/2)\pi, \quad n = 0, 1, \dots \quad (2.61)$$

or

$$E_n = \frac{(n + 1/2)^2 \pi^2 \hbar^2}{2ma^2}, \quad n = 0, 1, \dots \quad (2.62)$$

We have thus found an entire set of discrete energies for which the boundary condition is met, corresponding to the case with $A = 0, B \neq 0$, in which all the solutions are *even* functions of x .

We can also take $A \neq 0$. We then find that $\sin ka = 0$, so that $ka = 0, \pi, 2\pi, \dots$. This leads us to the *odd* solutions for which $A \neq 0$, $B = 0$, and

$$E_n = \frac{n^2 \pi^2 \hbar^2}{2ma^2}, \quad n = 1, 2, \dots \quad (2.63)$$

Again, we find an entire set of solutions corresponding to a particular parity. A general consequence of boundary conditions is this *restriction* to a set of solutions — not every value of the energy is permitted! Only certain values of the energy lead to solutions that satisfy both the differential equation *and* the boundary conditions: the *eigenvalues*.

But what if the potentials are not infinitely large? In the *finite square well* problem, we identify three different regions, as indicated in Figure 2.8:

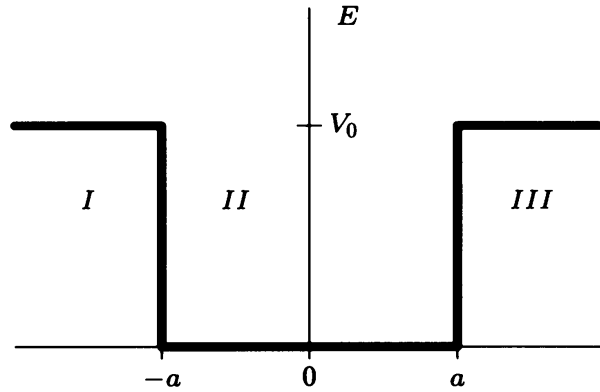


FIGURE 2.8 The finite square well.

We will only concern ourselves with states of the system that are localized, having energy less than V_0 . In region I, the Schrödinger equation is

$$\frac{d^2\psi}{dx^2} - \frac{2m}{\hbar^2}(V_0 - E)\psi = 0, \quad (2.64)$$

which has as a general solution

$$\psi_I(x) = Ce^{\beta x} + De^{-\beta x}, \quad \text{where } \beta = \sqrt{2m(V_0 - E)/\hbar^2}. \quad (2.65)$$

We might be tempted to require both C and D to be zero so that the wavefunction would be zero and there would be no possibility of finding the particle in region I. *But this is inconsistent with the experiment!* Sometimes, it's not easy being a physicist. One of the real surprises of quantum mechanics is that the wavefunction for a particle can be nonzero in places that classical mechanics would not allow the particle to be: region I is such a *classically forbidden*

region. What we do find, however, is that the farther into the classically forbidden region we look, the less likely it is to find the particle. That is, the wavefunction must *decrease* as it goes into the barrier. The correct boundary condition is then that D must identically vanish, else the probability would *increase* as the forbidden region was penetrated, contrary to the above discussion.

In region II the general solution is

$$\psi_{II} = A \sin \alpha x + B \cos \alpha x, \quad \alpha = \sqrt{\frac{2mE}{\hbar^2}}, \quad (2.66)$$

while in region III it must be

$$\psi_{III} = F e^{-\beta x} \quad (2.67)$$

to satisfy the boundary condition on forbidden regions. Furthermore, we are going to require that both $\psi(x)$ and $\psi'(x)$ be continuous — we don't expect to find a sudden change in where the particle is located. (Even in the infinite square well, the wavefunction was continuous across the boundaries. The discontinuity of the derivative was due to the infinite nature of the potential at that point.) At $x = -a$, this requires that

$$-A \sin \alpha a + B \cos \alpha a = C e^{-\beta a} \quad (2.68)$$

and

$$\alpha A \cos \alpha a + \alpha B \sin \alpha a = \beta C e^{-\beta a}, \quad (2.69)$$

while at $x = a$ we find

$$A \sin \alpha a + B \cos \alpha a = F e^{-\beta a} \quad (2.70)$$

and

$$\alpha A \cos \alpha a - \alpha B \sin \alpha a = -\beta F e^{-\beta a}. \quad (2.71)$$

After some algebra, we again find two cases, according to the parity of the solution:

$$\text{Even States: } A = 0, \quad B \neq 0, \quad C = F, \quad \alpha \tan \alpha a = \beta. \quad (2.72)$$

$$\text{Odd States: } A \neq 0, \quad B = 0, \quad C = -F, \quad \alpha \cot \alpha a = -\beta. \quad (2.73)$$

This is the result most often displayed in textbooks. We see that the original problem of finding the energies and wavefunctions of the finite square well

has evolved into the problem of finding the roots of a transcendental equation. And that's a problem we know how to solve!

Computing Strategy

When we're first presented with a substantial problem, such as this one, it is easy to become overwhelmed by its complexity. In the present case, the goal is to find the energies and wavefunctions of the finite square well, but we're not going to get there in one step. While we always want to remember where we're going, we need to break the original problem into several smaller chunks of a more manageable size, and solve them one at a time. This is the "modularization" we spoke of earlier, and we see a strong correlation between the process of solving the physical problem, and the writing of computer code that addresses a particular piece of the problem.

Our "main program" will initialize some variables, and then call one of the root-finding subroutines actually to find the root. It's in this main program that any issues of a global nature, issues that will pertain to the program as a whole, should be addressed. And in this project, we have such an issue: units.

In physics, we are accustomed to quantities such as mass having two attributes: their magnitude *and* their unit. Saying that a particle has mass 2 doesn't tell me much — is it 2 kilograms, or 2 metric tons? The computer, however, only knows magnitudes — it is up to the *computer* to keep the units straight. Sometimes this is easy, and sometimes not. In the current problem, we have attributes such as mass, distance, and energy to be considered. For macroscopic objects, expressing these attributes in kilograms, meters, and joules is natural, but these are extremely large units in which to express quantum mechanical entities. For example, the mass of the electron is about 9.11×10^{-31} kilograms — a perfectly good number, but rather small. It's best to use units that are natural to the problem at hand: for the square well problem, electron masses, Angstroms, and eV's are an appropriate set to use. Using $\hbar = 6.5821220 \times 10^{-16}$ eV·sec, we can then write

$$\hbar^2 = 7.6199682 m_e \text{ eV } \text{\AA}^2. \quad (2.74)$$

It's no accident that the numerical factor that appears here is on the order of one — in fact, that's the reason for this choice of units. These are not the conventional units of \hbar , but they work very nicely in this problem.

Let's imagine that we are to find the energy of the lowest state having

even parity. Then we would rewrite Equation (2.72) as

$$f(E) = \alpha \tan \alpha a - \beta = 0, \quad (2.75)$$

$$\text{where } \alpha = \sqrt{\frac{2mE}{\hbar^2}} \quad \text{and} \quad \beta = \sqrt{2m(V_0 - E)/\hbar^2}. \quad (2.76)$$

Now, the root-finders we've developed expect — actually, they demand! — that the root be bracketed. At this point, we have no idea where to start looking for the root, except that the root must lie between zero and the top of the well. An exhaustive search, simply calculating the function at several energies, say, every 0.1 eV up to 10 eV, could be conducted, but we would like to find the root with as few function evaluations as possible. A general strategy is to look for “special points” where the function has known forms. At $E = 0$, the function is easily evaluated as

$$f(0) = -\sqrt{2mV_0/\hbar^2}. \quad (2.77)$$

We also recognize that it goes to infinity as the argument of the tangent goes to $\pi/2$. In fact, the repetitive nature of the tangent function suggests to us that there might be several roots, one during each cycle of the function.

Use analytic results to establish limiting cases.

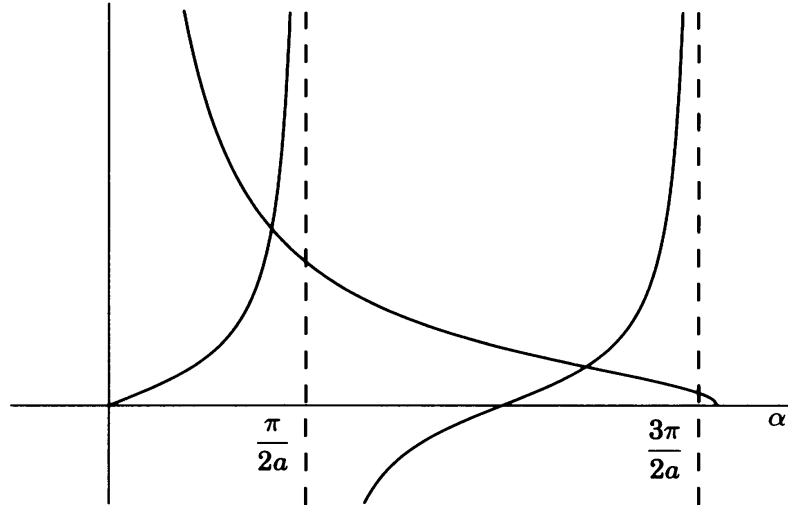


FIGURE 2.9 A plot of the functions $\tan \alpha a$ and β/α versus α .

In Figure 2.9, we've plotted $\tan \alpha a$ and β/α , for the case of $V_0 = 10$ eV, $a = 3$ Å, and $m = 1 m_e$, in which α ranges from zero to about 1.62 Å⁻¹. While

$\tan \alpha a$ contains about 1.5 cycles in this range, β/α decreases from infinity at $\alpha = 0$ to zero at

$$\alpha = \sqrt{2mV_0/\hbar^2}. \quad (2.78)$$

The figure clearly indicates a root lying in the range

$$0 < E < \frac{\pi^2 \hbar^2}{8ma^2} \approx 1.045 \text{ eV}. \quad (2.79)$$

(The one-dimensional square well *always* has a root, and hence at least one bound state always exists. This is *not* the case for three dimensions.)

Simple plots can help us visualize what's going on.

We now have rigorously bracketed the root during each cycle of the tangent function, but we're not quite ready to start. The difficulty is that we never want the computer actually to evaluate an infinity — which is exactly what the computer will try to do if instructed to evaluate the tangent at $\alpha a = \pi/2$. We could set the upper bound at some slightly smaller value, say, 0.999999 times the upper limit of the energy. But if this value is too small, the root would not be bounded. Instead of trying to “patch the code” to make it work, let's see if we can find the true origin of the difficulty.

When we imposed boundary conditions, we found that for even states

$$B \cos \alpha a = C e^{-\beta a} \quad (2.80)$$

and

$$\alpha B \sin \alpha a = \beta C e^{-\beta a}. \quad (2.81)$$

We then wrote this requirement as

$$\alpha \tan \alpha a = \beta, \quad (2.82)$$

but it could just as easily have been written as

$$\beta \cos \alpha a = \alpha \sin \alpha a. \quad (2.83)$$

In fact, Equation (2.83) is just a statement of the matching condition, with common terms eliminated. To obtain Equation (2.82) from (2.83) we had to divide by $\cos \alpha a$ — our infinity problem originates here, when $\alpha a = \pi$ and

we're dividing by zero! We can totally avoid the infinity problem, and simultaneously improve the correspondence between computer code and the mathematical boundary conditions, by replacing the transcendental equations (2.72) and (2.73) by

$$\textbf{Even States: } A = 0, \quad B \neq 0, \quad C = F, \quad \beta \cos \alpha a = \alpha \sin \alpha a. \quad (2.84)$$

$$\text{Odd States: } A \neq 0, \quad B = 0, \quad C = -F, \quad \alpha \cos \alpha a = -\beta \sin \alpha a. \quad (2.85)$$

In analogy to Figure 2.9, we could now plot $\beta \cos \alpha a$ and $\alpha \sin \alpha a$ versus α . The curves would cross at exactly the same points as do β/α and $\tan \alpha a$, but would be preferable in the sense that they have no singularities in them. However, having the capability of the computer to plot for us creates many options. For instance, there's no longer any reason to be using α as the independent variable — while α is a convenient variable for humans, the computer would just as soon use the energy directly! This facilitates a more straightforward approach to the solution of the problem before us.

All we have left to do is to code the FUNCTION itself. It's no accident that this is the last step in the process of developing a computer program to solve our physical problem — although clearly important to the overall goal, it's at the end of the logical chain, not the beginning. For the even parity solutions, the function might be coded something like this:

[illegible]

```

      Double Precision E, a, V0, Mass, h_bar_SQ, alpha, beta
*
* Specify constants:
*
      PARAMETER( a = ? , V0 = ?)
      PARAMETER( Mass = ?, h_bar_SQ = 7.6199682d0)
*
* Evaluate the function and return.
*
      alpha = sqrt( 2*Mass* E / h_bar_SQ )
      beta = sqrt( 2*Mass(V0-E)/ h_bar_SQ )
      even = beta * cos(alpha*a) - alpha * sin(alpha*a)
      end

```

We've used the `PARAMETER` statement to specify the constants so that they can't be accidentally changed within the program.

❏ EXERCISE 2.11

Plot the function

$$f(E) = \beta \cos \alpha a - \alpha \sin \alpha a$$

as a function of energy.

All that's left is to solve the original problem!

❏ EXERCISE 2.12

Find the lowest even and lowest odd solutions to the square well problem, with $a = 3 \text{ \AA}$, $m = 1 m_e$, and $V_0 = 10 \text{ eV}$. Plot the potential and the wavefunctions associated with these eigenvalues. (The convention is to align the zero baseline of the wavefunction with the energy eigenvalue.)

It's quite common in physics that the solution to one problem just leads to more questions. You've now found the lowest energies of the square well for a particular width parameter a , but doesn't that make you wonder how the energies would *change* with a ? Or with V_0 ? To answer these questions, you need to transform the program you now have into a subroutine, remove V_0 and a from the parameter list, and call your new subroutine from the main program with different values of V_0 and a .

EXERCISE 2.13

Investigate the dependence of the lowest two eigenvalues (energies) of the square well upon V_0 (with a fixed at 3 \AA) and a (with V_0 fixed at 10 eV), and plot your results. Since you will want V_0 to extend to infinity, you will probably want to plot the energies versus V_0^{-1} , and versus a . What is the smallest width that will support a bound state?

And what would happen if you had not one square well, but two, as in Figure 2.10? Of course, you would need to develop different solutions than the ones used here, with five regions of interest instead of three, but the process is the same. And the double square well exhibits some interesting behavior, not seen in the single square well problem. How do the lowest energies of even and odd parity change as the distance between the wells change? What do the eigenfunctions look like? From physical reasoning, how must the lowest eigenvalues and their wavefunctions behave in the limit that the distance between the wells vanishes?

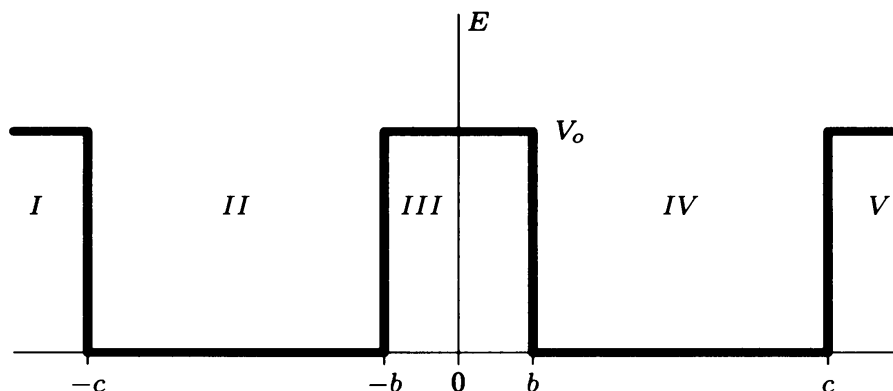


FIGURE 2.10 The double square well.

EXERCISE 2.14

Consider the double square well, with a single well described with the parameters $a = 3 \text{ \AA}$, $m = 1 m_e$, and $V_0 = 10 \text{ eV}$. If the two wells are far apart, there are two energies very near the energy of a single well. Why? Investigate the dependence of these energies as the wells are brought nearer to one another.

References

Root finding is a standard topic of numerical analysis and is discussed in many such texts, including

Anthony Ralston, *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1965.

Richard L. Burden and J. Douglas Faires, *Numerical Analysis*, Prindle, Weber & Schmidt, Boston, 1985.

Curtis F. Gerald and Patrick O. Wheatley, *Applied Numerical Analysis*, Addison-Wesley, Reading, Massachusetts, 1989.

Although somewhat dated, the following text is particularly commendable with regard to the author's philosophy of computation:

Forman S. Acton, *Numerical Methods That Work*, Harper & Row, New York, 1970.

The modifications of Brent and Müller are discussed in

J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980.

Chapter 3:

Interpolation and

Approximation

In the last chapter, we noted that an *approximation* to a function was useful in finding its root, even though we had the exact function at our disposal. Perhaps a more common circumstance is that we don't know the exact function, but build our knowledge of it as we acquire more information about it, one point at a time. In either case, it's important for us to incorporate the information we have into an approximation that is useful to us. Presumably, as we gather more information, our approximation becomes better.

In this chapter, several ways to approximate a function and its derivatives are investigated. With *interpolation*, an approximating polynomial is found that exactly describes the function being approximated at a set of specified points. Lagrange and Hermite interpolation are discussed, and the use of cubic splines is developed. Application of Taylor's series methods to the evaluation of derivatives is also discussed, as is the important technique of Richardson extrapolation. *Curve fitting*, which approximates a function in a general sense, without being constrained to agree with the function at every point, is discussed — the method of least squares is developed in this regard. Along the way, a need to solve sets of linear equations is encountered and so a method to solve them is developed. In passing, we note that functions can be expressed in terms of other functions, and that sets of orthogonal functions are particularly convenient in this regard. Finally, the method of least squares fitting using non-polynomial functions is developed, leading us to consider minimization methods in multiple dimensions.

Lagrange Interpolation

While Taylor's series can be used to approximate a function at x if the function

and its derivatives are known at some point, a method due to Lagrange can be used to approximate a function if only the function is known, although it must be known at several points. We can derive *Lagrange's interpolating polynomial* $p(x)$ from a Taylor's series by expressing the function at x_1 and x_2 in terms of the function and its derivatives at x ,

$$\begin{aligned} f(x_1) &= f(x) + (x_1 - x)f'(x) + \cdots, \\ f(x_2) &= f(x) + (x_2 - x)f'(x) + \cdots. \end{aligned} \quad (3.1)$$

We would like to truncate the series and retain only the first two terms. But in so doing, the *equality* would be compromised. However, we can introduce *approximations* to the function and its derivative such that an equality is retained. That is, we introduce the new function $p(x)$ and its derivative such that

$$f(x_1) = p(x) + (x_1 - x)p'(x)$$

and

$$f(x_2) = p(x) + (x_2 - x)p'(x). \quad (3.2)$$

Clearly, $p(x)$ is equal to $f(x)$ at x_1 and x_2 , and is perhaps a reasonable approximation in their vicinity. We then have two equations in the two unknowns $p(x)$ and $p'(x)$; solving for $p(x)$, we find

$$p(x) = \frac{x - x_2}{x_1 - x_2} f(x_1) + \frac{x - x_1}{x_2 - x_1} f(x_2), \quad (3.3)$$

a linear function in x . This is nothing more than the equation of the line passing through the points $[x_1, f(x_1)]$ and $[x_2, f(x_2)]$, and could have been found by other means. In fact, we used this equation in Chapter 2 in developing the method of false position.

But the *form* of Equation (3.3) is very convenient, and interesting. For example, it says that the contribution of $f(x_2)$ to the approximation is weighted by a given factor, $(x - x_1)/(x_2 - x_1)$, which depends upon the distance x is away from x_1 . As x varies between x_1 and x_2 , this factor increases (linearly) from 0 to 1, so that the importance of $f(x_2)$ to the approximation varies from “irrelevant” to “sole contributor.” At the same time, the factor multiplying $f(x_1)$ behaves in a complementary way, decreasing linearly as x varies between x_1 to x_2 .

A higher order approximation can easily be obtained by retaining more terms in the series expansion. Of course, if another term is retained, the function must be known at an additional point as well. For example, the

three equations

$$\begin{aligned} f(x_1) &= f(x) + (x_1 - x)f'(x) + \frac{(x_1 - x)^2}{2}f''(x) + \cdots, \\ f(x_2) &= f(x) + (x_2 - x)f'(x) + \frac{(x_2 - x)^2}{2}f''(x) + \cdots, \\ f(x_3) &= f(x) + (x_3 - x)f'(x) + \frac{(x_3 - x)^2}{2}f''(x) + \cdots, \end{aligned} \quad (3.4)$$

can be truncated, the functions replaced by their approximations, and the equations solved to yield the quadratic interpolating polynomial

$$\begin{aligned} p(x) &= \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}f(x_1) + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}f(x_2) \\ &\quad + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}f(x_3). \end{aligned} \quad (3.5)$$

Again, we see that $p(x_j) = f(x_j)$. Earlier, we obtained $p(x)$ by writing $p(x) = ax^2 + bx + c$ and determining a , b , and c by requiring that the approximation be equal to the function at three points. Since there is one and only one quadratic function passing through any three points, the interpolating polynomial of Equation (3.5) must be identical to the one we found earlier, although it certainly doesn't look the same.

Equations (3.3) and (3.5) suggest that a general interpolating polynomial of order $(n - 1)$ might be written as

$$p(x) = \sum_{j=1}^n l_{j,n}(x) f(x_j), \quad (3.6)$$

where the function $f(x)$ is known at the n points x_j and

$$l_{j,n}(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_1)(x_j - x_2) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)} \quad (3.7)$$

is the coefficient of $f(x_j)$. Note that

$$l_{j,n}(x_i) = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

a relation that is compactly expressed in terms of the Kronecker delta,

$$l_{j,n}(x_i) = \delta_{i,j}. \quad (3.8)$$

This approximation to $f(x)$ is known as the Lagrange interpolating polynomial. It can be extended to more points — later we will argue that this is not a good practice, however.

Once upon a time — before the widespread availability of computers — interpolation was an extremely important topic. Values of special functions of interest to mathematical physics, such as Bessel functions, were laboriously calculated and entered into tables. When a particular value of the function was desired, interpolation was used on the tabulated values. Sophisticated, specialized methods were developed to perform interpolation of tabular data in an accurate and efficient manner.

Today, it is more likely to have a function evaluated as needed, rather than interpolated, so that the importance of interpolation with respect to tabular data is somewhat diminished. But the topic of interpolation is still important, because sometimes the data are simply not available at the points of interest. Certainly, this often happens with experimental data. (Depending upon the circumstances, there might be more appropriate ways to analyze experimental data, however.) Interpolation also provides a theoretical basis for the discussion of other topics, such as differentiation and integration.

The Airy Pattern

When light enters a telescope, only that which falls upon the lens (or mirror) is available to the astronomer. The situation is equivalent to an infinitely large screen, with a circular hole the size of the lens (or mirror) cut in it. And we all know that when light passes through an aperture like that, it suffers diffraction. For a circular aperture, the resulting pattern of light and dark rings is known as the Airy pattern, named after Sir George Airy, the Astronomer Royal, who first described it in 1835. This diffraction distorts the image of even a point source of light, although this distortion is usually small compared to other sources of distortion. Only in the best of telescopes have these other sources been eliminated to the point that the quality of the image is *diffraction limited*. In this instance, the intensity of the light is described by the function

$$I = I_0 \left[\frac{2J_1(\rho)}{\rho} \right]^2, \quad (3.9)$$

where I_0 is intensity of the incident light and $J_1(\rho)$ is the Bessel function of order 1. (The center of the image is at $\rho = 0$.) The Bessel function has many interesting characteristics, and is often studied in mathematical physics courses and discussed at length in textbooks. A few values of the Bessel function are

listed in Table 3.1.

TABLE 3.1 Bessel Functions

ρ	$J_0(\rho)$	$J_1(\rho)$	$J_2(\rho)$
0.0	1.00000 00000	0.00000 00000	0.00000 00000
1.0	0.76519 76866	0.44005 05857	0.11490 34849
2.0	0.22389 07791	0.57672 48078	0.35283 40286
3.0	-0.26005 19549	0.33905 89585	0.48609 12606
4.0	-0.39714 98099	-0.06604 33280	0.36412 81459
5.0	-0.17759 67713	-0.32757 91376	0.04656 51163
6.0	0.15064 52573	-0.27668 38581	-0.24287 32100
7.0	0.30007 92705	-0.00468 28235	-0.30141 72201
8.0	0.17165 08071	0.23463 63469	-0.11299 17204
9.0	-0.09033 36112	0.24531 17866	0.14484 73415
10.0	-0.24593 57645	0.04347 27462	0.25463 03137

EXERCISE 3.1

Using pencil and paper, estimate $J_1(5.5)$ by linear, quadratic, and cubic interpolation.

You probably found the computations in the Exercise to be tedious — certainly, they are for high order approximations. One of the advantages of the Lagrange approach is that its coding is very straightforward. The crucial fragment of the code, corresponding to Equations (3.3) and (3.5), might look like

```

*
* Code fragment for Lagrange interpolation using N points.
* The approximation P is required at XX, using the
* tabulated values X(j) and F(j).
*
      P = 0.d0
      DO j = 1, N
*
* Evaluate the j-th coefficient
*
          Lj = 1.d0
          DO k = 1, N
              IF(j .ne. k) THEN
                  Lj = Lj * ( xx-x(k) )/( x(j)-x(k) )
              ENDIF
          
```

```

      END DO
*
*   Add contribution of j-th term to the polynomial
*
      P = P + Lj * F(j)
    END DO
    ...

```

We should point out that products need to be initialized to 1, just as sums need to be initialized to 0. This fragment has one major flaw — what happens if any two of the x_i are the same? While this doesn't happen in the present example, it's a possibility that should be addressed in a general interpolation program.

In the above exercise, which ρ values did you use in your linear interpolation? Nothing we've said so far would prevent you from having used $\rho = 0$ and 1, but you probably used $\rho = 5$ and 6, didn't you? You know that at $\rho = 5$ the approximating function is exact. As you move away from 5, you would expect there to be a difference to develop between the exact function and the approximation, but this difference (i.e., error) must become zero at $\rho = 6$. So if you use tabulated values at points that surround the point at which you are interested, the error will be kept to a minimum.

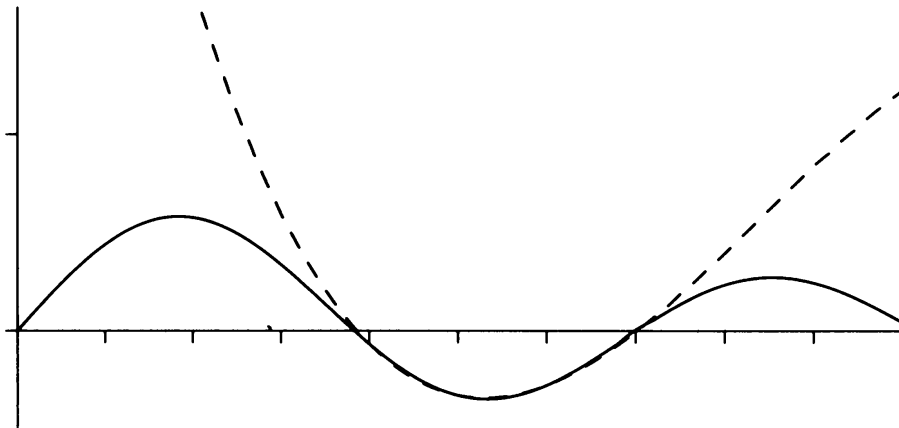


FIGURE 3.1 The Bessel function $J_1(\rho)$ and the cubic Lagrange polynomial approximation to it, using the tabular data at $\rho = 4, 5, 6$, and 7.

The converse is particularly illuminating. Using the interpolating polynomial to *extrapolate* to the region exterior to the points sampled can

lead to disastrous results, as seen in Figure 3.1. Here the cubic interpolation formula derived from sampling ρ at 4, 5, 6, and 7 is plotted versus the true $J_1(\rho)$. Between 5 and 6, and even between 4 and 7, the approximation is very good. But there's only so much flexibility in a cubic function, and so the interpolating polynomial must eventually fail as we use it outside of that region. An extreme example is in fitting a periodic function, such as a cosine, with a polynomial. An n -th degree polynomial only has $(n - 1)$ extrema, and so cannot possibly follow the infinite oscillations of a periodic function.

One would think that by increasing the order of the polynomial, we could improve the fit. And to some extent, we can. Over some specific region, say, $5 \leq \rho \leq 6$, a cubic fit will almost surely be better than a linear one. But as we consider higher order approximations, we see less of an improvement. The reason, of course, is that the higher order polynomials require more points in their determination. These points are *farther* from the region of interest and so do little to enhance the quality of the approximation.

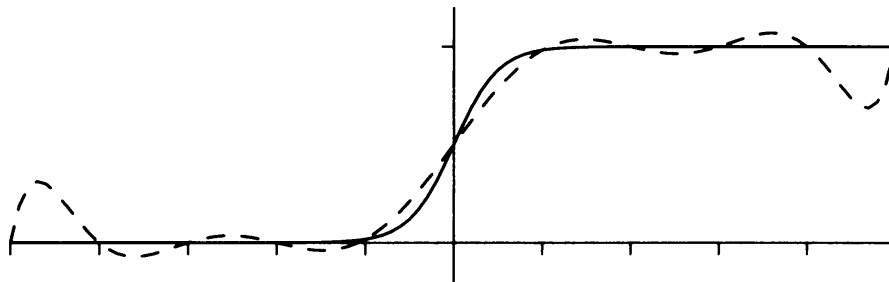


FIGURE 3.2 An example of the failure of a high order interpolating polynomial.

And there is another factor, particularly for evenly spaced data — a high order interpolating polynomial necessarily has “a lot of wiggle in it,” even if it doesn’t belong! For example, consider the function

$$f(x) = \frac{1 + \tanh 2\alpha x}{2}. \quad (3.10)$$

This function has some interesting characteristics. As $\alpha \rightarrow \infty$, $f(x)$ becomes zero for $x < 0$ and unity for $x > 0$. That is, it tends toward the Heaviside step function! For $\alpha = 10$, $f(x)$ is plotted in Figure 3.2 on the interval $-1 \leq x \leq 1$. The curve is smooth and well-behaved, although it does possess a “kink” at $x = 0$ — one that tends towards a discontinuity as $\alpha \rightarrow \infty$. We’ve also plotted the Lagrange interpolating polynomial obtained from using 11 evenly spaced

points in this interval. Clearly, the approximation is failing, particularly at the ends of the interval.

EXERCISE 3.2

Use a 21-point interpolating polynomial to approximate the function in Equation (3.10). Plot the approximation and the function, as in Figure 3.2. Does the higher order approximation help?

Experience suggests that the high order approximations aren't very useful. Generally speaking, a cubic approximation, with the points chosen so as to surround the desired point of evaluation, usually works pretty well. In developing an appropriate computer routine, you need to develop an "automatic" way of surrounding the point, and make a special provision for the two ends of the table. For ease-of-use, the code should be packaged as a FUNCTION.

EXERCISE 3.3

Using the FUNCTION as described above to interpolate on the table of Bessel functions, write a computer program to evaluate the relative intensity I/I_0 as a function of ρ across the Airy diffraction pattern, and plot it. To generate a "pleasing" display, you should evaluate the intensity at 0.1 increments in ρ .

Hermite Interpolation

Earlier we commented that having distant information about a function was not of great help in approximating it on a specific region. But if we could have more information about the function *near* the region of evaluation, surely that must be of help! It sometimes happens that information about the derivative of the function, as well as the function itself, is available to us. If it is available, then we should certainly use it!

In the present example, we have a table of Bessel functions of various orders at certain tabulated points. But the Bessel functions have some special properties. (Again, the interested reader is encouraged to peruse a suitable reference.) For example,

$$J'_0(x) = -J_1(x), \quad (3.11)$$

and

$$J'_n(x) = \frac{J_{n-1}(x) - J_{n+1}(x)}{2}. \quad (3.12)$$

So we do have information about the function and its derivative! For this general case, we propose the interpolation polynomial

$$p(x) = ax^3 + bx^2 + cx + d \quad (3.13)$$

and determine a , b , c , and d by requiring that

$$\begin{aligned} p(x_1) &= f(x_1), & p(x_2) &= f(x_2), \\ p'(x_1) &= f'(x_1), & \text{and} & & p'(x_2) &= f'(x_2). \end{aligned} \quad (3.14)$$

This interpolating polynomial will be continuous, as was the Lagrange interpolating polynomial, and its first derivative will also be continuous! With some effort, we can determine the appropriate coefficients and find

$$\begin{aligned} p(x) &= \frac{(1 - 2(x - x_1))(x - x_2)^2}{(x_1 - x_2)^2} f(x_1) + \frac{(1 - 2(x - x_2))(x - x_1)^2}{(x_1 - x_2)^2} f(x_2) \\ &+ \frac{(x - x_1)(x - x_2)^2}{(x_1 - x_2)^2} f'(x_1) + \frac{(x - x_2)(x - x_1)^2}{(x_1 - x_2)^2} f'(x_2). \end{aligned} \quad (3.15)$$

In this example, we've only considered Hermite cubic interpolation, but it should be clear that we could devise other polynomials, depending upon the information we have available. For example, if we knew the function at n points and the derivative at r points, we could construct a polynomial of order $n+r-1$ satisfying the $n+r$ conditions. The more frequent circumstance is that we know the function and its derivative at n points. Using all the information at our disposal, we write the general Hermite interpolating polynomial as

$$p(x) = \sum_{j=1}^n h_{j,n}(x) f(x_j) + \sum_{j=1}^n \bar{h}_{j,n}(x) f'(x_j), \quad (3.16)$$

where h and \bar{h} are (as yet) undetermined polynomials. With some effort, we can find that

$$h_{j,n} = [1 - 2(x - x_j)l'_{j,n}(x_j)] l_{j,n}^2(x) \quad (3.17)$$

and

$$\bar{h}_{j,n}(x) = (x - x_j) l_{j,n}^2(x), \quad (3.18)$$

where the $l_{j,n}(x)$ were defined in association with the Lagrange polynomial. The Hermite polynomials are termed osculating, by the way, since they are constructed so as to just “kiss” at the points x_j .

EXERCISE 3.4

Using Hermite interpolation, evaluate the relative intensity I/I_0 , and compare to the results of the previous exercise. In conjunction with a root-finder, determine the ρ -value for which the intensity is zero, i.e., find the location of the first fringe in the diffraction pattern.

It is also possible to construct approximations using higher derivatives. The Bessel functions, for example, satisfy the differential equation

$$x^2 J_n'' + x J_n' + (x^2 - n^2) J_n = 0, \quad (3.19)$$

so that if we have a table of J_n 's, we can find both J_n' and J_n'' . A 5th-order approximating polynomial could then be determined, for example, passing through two points. Since this information is “local” to the region being approximated, these high order approximations do not suffer the unstable behavior of the high order Lagrange polynomials.

Cubic Splines

If derivatives are available, then Hermite interpolation can — and probably should — be used. That will guarantee that the function and its first derivative will be continuous. More often than not, however, the derivatives are not available. While the Lagrange interpolating polynomial can certainly be used, it has one very undesirable characteristic: its derivative is not continuous. Let's imagine that we're using the points x_1 , x_2 , x_3 , and x_4 and interpolating in the region $x_2 \leq x \leq x_3$. As x varies from x_2 to x_3 , the function and its derivatives vary smoothly. But as x increases beyond x_3 , we change the points used in the interpolation in order to keep x “surrounded.” This shifting of interpolation points is done for a good reason, as discussed earlier, but look at the consequence: since we now have a different set of interpolation points, we have a different approximating polynomial. Although the *function* will be continuous, the *derivatives* will suffer a discontinuous change. Not a particularly attractive feature.

It would be desirable to have an approximating function that had continuous derivatives. And in fact, we can construct such a function. Let's define $p(x)$ as the cubic interpolating function used in the region $x_j \leq x \leq x_{j+1}$, which can be written as

$$p(x) = a_j(x - x_j)^3 + b_j(x - x_j)^2 + c_j(x - x_j) + d_j. \quad (3.20)$$

Requiring this approximation to be exact at $x = x_j$ gives us

$$p(x_j) = f(x_j) = d_j, \quad (3.21)$$

This approximation should also be exact at $x = x_{j+1}$, so that

$$p_{j+1} = a_j h_j^3 + b_j h_j^2 + c_j h_j + p_j, \quad (3.22)$$

where we've introduced the notation

$$p_j = p(x_j) \quad \text{and} \quad h_j = x_{j+1} - x_j. \quad (3.23)$$

The derivatives of our cubic approximation are

$$p'(x) = 3a_j(x - x_j)^2 + 2b_j(x - x_j) + c_j \quad (3.24)$$

and

$$p''(x) = 6a_j(x - x_j) + 2b_j. \quad (3.25)$$

For the second derivative at $x = x_j$ we have

$$p''(x_j) = p_j'' = 2b_j \quad (3.26)$$

so that

$$b_j = \frac{p_j''}{2}, \quad (3.27)$$

while at $x = x_{j+1}$ we have

$$p_{j+1}'' = 6a_j h_j + 2b_j \quad (3.28)$$

and hence

$$a_j = \frac{1}{6} \frac{p_{j+1}'' - p_j''}{h_j}. \quad (3.29)$$

From Equation (3.22), we then find that

$$c_j = \frac{p_{j+1} - p_j}{h_j} - \frac{h_j p_{j+1}'' + 2h_j p_j''}{6}. \quad (3.30)$$

With the coefficients of the polynomial known, at least in terms of the p_j'' , we can write

$$\begin{aligned} p(x) = p_j + & \left[\frac{p_{j+1} - p_j}{h_j} - \frac{h_j p_{j+1}''}{6} - \frac{h_j p_j''}{3} \right] (x - x_j) + \frac{p_j''}{2} (x - x_j)^2 \\ & + \frac{p_{j+1}'' - p_j''}{6h_j} (x - x_j)^3, \quad x_j \leq x \leq x_{j+1}, \end{aligned} \quad (3.31)$$

and

$$p'(x) = \frac{p_{j+1} - p_j}{h_j} - \frac{h_j p''_{j+1}}{6} - \frac{h_j p''_j}{3} + p''_j(x - x_j) + \frac{p''_{j+1} - p''_j}{2h_j}(x - x_j)^2, \quad x_j \leq x \leq x_{j+1}. \quad (3.32)$$

These expressions tell us that the function and its derivative can be approximated from the p_j and the p''_j . Of course, we already know the p_j . To determine the p''_j , we consider the derivative in the *previous* interval. Replacing j by $j - 1$ in Equation (3.32), we have

$$p'(x) = \frac{p_j - p_{j-1}}{h_{j-1}} - \frac{h_{j-1} p''_j}{6} - \frac{h_{j-1} p''_{j-1}}{3} + p''_{j-1}(x - x_{j-1}) + \frac{p''_j - p''_{j-1}}{2h_{j-1}}(x - x_{j-1})^2, \quad x_{j-1} \leq x \leq x_j. \quad (3.33)$$

We now require that Equations (3.33) and (3.32) yield exactly the same value of the derivative at $x = x_j$, so that

$$\begin{aligned} \frac{p_j - p_{j-1}}{h_{j-1}} - \frac{h_{j-1} p''_j}{6} - \frac{h_{j-1} p''_{j-1}}{3} + p''_{j-1}h_j + \frac{p''_j - p''_{j-1}}{2h_{j-1}}h_{j-1}^2 \\ = \frac{p_{j+1} - p_j}{h_j} - \frac{h_j p''_{j+1}}{6} - \frac{h_j p''_j}{3}. \end{aligned} \quad (3.34)$$

Moving the (unknown) p''_j to the left side of the equation and the (known) p_j to the right, we find

$$\begin{aligned} h_{j-1} p''_{j-1} + (2h_j + 2h_{j-1})p''_j + h_j p''_{j+1} \\ = 6 \left(\frac{p_{j+1} - p_j}{h_j} - \frac{p_j - p_{j-1}}{h_{j-1}} \right), \quad j = 2, \dots, n-1. \end{aligned} \quad (3.35)$$

This gives us $(n - 2)$ equations for the n unknown p''_j — we need two more equations to determine a unique solution. These additional equations may come from specifying the derivative at the endpoints, i.e., at $x = x_1$ and x_n . From Equation (3.32), we find

$$2h_1 p''_1 + h_1 p''_2 = 6 \frac{p_2 - p_1}{h_1} - 6p'_1, \quad (3.36)$$

while from Equation (3.33) we find

$$h_{n-1}p''_{n-1} + 2h_{n-1}p''_n = -6\frac{p_n - p_{n-1}}{h_{n-1}} + p'_n. \quad (3.37)$$

All these equations can be written in matrix form as

$$\begin{bmatrix} 2h_1 & h_1 & & & \\ h_1 & 2(h_1 + h_2) & h_2 & & \\ & h_2 & 2(h_2 + h_3) & h_3 & \\ & & & \ddots & \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ & & & & h_{n-1} & 2h_{n-1} \end{bmatrix} \begin{bmatrix} p''_1 \\ p''_2 \\ p''_3 \\ \vdots \\ p''_{n-1} \\ p''_n \end{bmatrix} = \begin{bmatrix} 6\frac{p_2 - p_1}{h_1} - 6p'_1 \\ 6\frac{p_3 - p_2}{h_2} - 6\frac{p_2 - p_1}{h_1} \\ 6\frac{p_4 - p_3}{h_3} - 6\frac{p_3 - p_2}{h_2} \\ \vdots \\ 6\frac{p_n - p_{n-1}}{h_{n-1}} - 6\frac{p_{n-1} - p_{n-2}}{h_{n-1}} \\ -6\frac{p_n - p_{n-1}}{h_{n-1}} + 6p'_n \end{bmatrix}. \quad (3.38)$$

Is it obvious why the matrix is termed *tridiagonal*?

The derivatives at the endpoints are not always known, however. The most common recourse in this instance is to use the so-called *natural spline*, obtained by setting the second derivatives to zero at $x = x_1$ and $x = x_n$. This forces the approximating function to be linear at the limits of the interpolating region. In this case the equations to be solved are

$$\begin{bmatrix} 1 & & & & \\ & 2(h_1 + h_2) & h_2 & & \\ & h_2 & 2(h_2 + h_3) & h_3 & \\ & & & \ddots & \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & 1 \end{bmatrix} \begin{bmatrix} p''_1 \\ p''_2 \\ p''_3 \\ \vdots \\ p''_{n-1} \\ p''_n \end{bmatrix}$$

$$= \begin{bmatrix} 0 & & & \\ 6\frac{p_3 - p_2}{h_2} & -6\frac{p_2 - p_1}{h_1} & & \\ 6\frac{p_4 - p_3}{h_3} & -6\frac{p_3 - p_2}{h_2} & & \\ \vdots & \vdots & \vdots & \vdots \\ 6\frac{p_n - p_{n-1}}{h_{n-1}} & -6\frac{p_{n-1} - p_{n-2}}{h_{n-1}} & & \\ 0 & & & \end{bmatrix}. \quad (3.39)$$

Again, the equations are of a tridiagonal form. This is the simplest form that allows for a second derivative. Since many of the important equations of physics are second-order differential equations, tridiagonal systems like these arise frequently in computational physics. Before proceeding with our problem of determining cubic splines, let's investigate the general solution to systems of equations of this form.

Tridiagonal Linear Systems

Linear algebraic systems are quite common in applied problems. In fact, since they are relatively easy to solve, one way of attacking a difficult problem is to find some way to write it as a linear one. In the general case, where a solution to an arbitrary set of simultaneous linear equations is being sought, Gaussian elimination with partial pivoting is a common method of solution and will be discussed later in this chapter. For tridiagonal systems, Gaussian elimination takes on a particularly simple form, which we discuss here.

Let's write a general tridiagonal system as

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_{n-1} \\ r_n \end{bmatrix}. \quad (3.40)$$

The b_j , $j = 1, \dots, n$ lie on the main diagonal. On the subdiagonal lie the a_j , for which j ranges from 2 to n , while the c_j lie above the main diagonal, with $1 \leq j \leq n - 1$. Alternatively, we can write Equation (3.40) as a set of

simultaneous equations,

$$\begin{array}{rclcl}
 b_1x_1 & +c_1x_2 & & & = & r_1 \\
 a_2x_1 & +b_2x_2 & +c_2x_3 & & = & r_2 \\
 & a_3x_2 & +b_3x_3 & +c_3x_4 & = & r_3 \\
 & & & & \vdots & \\
 & a_{n-1}x_{n-2} & +b_{n-1}x_{n-1} & +c_{n-1}x_n & = & r_{n-1} \\
 & & a_nx_{n-1} & +b_nx_n & = & r_n
 \end{array} \quad (3.41)$$

The general way to solve such sets of equations is to combine the equations in such a way as to eliminate some of the variables. Let's see if we can eliminate x_1 from the first two equations. Multiply the first equation by a_2/b_1 , and subtract it from the second, and use this new equation in place of the original second equation to obtain the set

$$\begin{array}{rclcl}
 b_1x_1 & +c_1x_2 & & & = & r_1 \\
 (b_2 - \frac{a_2}{b_1}c_1)x_2 & +c_2x_3 & & & = & r_2 - \frac{a_2}{b_1}r_1 \\
 & a_3x_2 & +b_3x_3 & +c_3x_4 & = & r_3 \\
 & & & & \vdots & \\
 & a_{n-1}x_{n-2} & +b_{n-1}x_{n-1} & +c_{n-1}x_n & = & r_{n-1} \\
 & & a_nx_{n-1} & +b_nx_n & = & r_n
 \end{array} \quad (3.42)$$

To simplify the notation, define

$$\beta_1 = b_1, \quad \rho_1 = r_1, \quad (3.43)$$

and

$$\beta_2 = b_2 - \frac{a_2}{\beta_1}c_1, \quad \rho_2 = r_2 - \frac{a_2}{\beta_1}\rho_1, \quad (3.44)$$

thus obtaining

$$\begin{array}{rclcl}
 \beta_1x_1 & +c_1x_2 & & & = & \rho_1 \\
 & \beta_2x_2 & +c_2x_3 & & = & \rho_2 \\
 & a_3x_2 & +b_3x_3 & +c_3x_4 & = & r_3 \\
 & & & & \vdots & \\
 & a_{n-1}x_{n-2} & +b_{n-1}x_{n-1} & +c_{n-1}x_n & = & r_{n-1} \\
 & & a_nx_{n-1} & +b_nx_n & = & r_n
 \end{array} \quad (3.45)$$

We can now proceed to eliminate x_2 — multiplying the second equation by

a_3/β_2 and subtracting it from the third yields

$$\begin{array}{rcl}
 \beta_1 x_1 & +c_1 x_2 & = \rho_1 \\
 & \beta_2 x_2 & +c_2 x_3 = \rho_2 \\
 & & +\beta_3 x_3 +c_3 x_4 = \rho_3 \\
 & & \vdots \\
 & a_{n-1} x_{n-2} & +b_{n-1} x_{n-1} +c_{n-1} x_n = r_{n-1} \\
 & & a_n x_{n-1} +b_n x_n = r_n
 \end{array}, \quad (3.46)$$

where we've defined

$$\beta_3 = b_3 - \frac{a_3}{\beta_2} c_2, \quad \rho_3 = r_3 - \frac{a_3}{\beta_2} \rho_2. \quad (3.47)$$

Clearly, there's a pattern developing here. After $n - 1$ such steps, we arrive at the set of equations

$$\begin{array}{rcl}
 \beta_1 x_1 & +c_1 x_2 & = \rho_1 \\
 & \beta_2 x_2 & +c_2 x_3 = \rho_2 \\
 & & +\beta_3 x_3 +c_3 x_4 = \rho_3 \\
 & & \vdots \\
 & \beta_{n-1} x_{n-1} & +c_{n-1} x_n = \rho_{n-1} \\
 & & \beta_n x_n = \rho_n
 \end{array}, \quad (3.48)$$

where we've defined

$$\beta_j = b_j - \frac{a_j}{\beta_{j-1}} c_{j-1} \quad \text{and} \quad \rho_j = r_j - \frac{a_j}{\beta_{j-1}} \rho_{j-1}, \quad j = 2, \dots, n. \quad (3.49)$$

This set of equations can now be solved by “back substitution.” From the last of the equations, we have

$$x_n = \rho_n / \beta_n, \quad (3.50)$$

which can then be substituted into the previous equation to yield

$$x_{n-1} = (\rho_{n-1} - c_{n-1} x_n) / \beta_{n-1}, \quad (3.51)$$

and so on. Since all the previous equations have the same form, we are lead to the general result

$$x_{n-j} = (\rho_{n-j} - c_{n-j} x_{n-j+1}) / \beta_{n-j}, \quad j = 1, \dots, n-1, \quad (3.52)$$

the solution to our original problem!

A subroutine to implement the solution we've developed can easily be written. In fact, all that needs to be done is to determine the β_j 's and ρ_j 's from Equations (3.43) and (3.49), and then back substitute according to Equations (3.50) and (3.52) to determine the x_j 's. A suitable code might look like

```

      Subroutine TriSolve(A, B, C, X, R, n)
* This subroutine solves the tridiagonal set of equations
*
* / b(1)  c(1)                \ / x(1)\ / r(1)\
* | a(2)  b(2)  c(2)          | | x(2) | | r(2) |
* |      a(3)  b(3)  c(3)      | | x(3) | | r(3) |
* |                        ... | | .... | = | .... |
* |          a(n-1) b(n-1) c(n-1) | | x(n-1) | | r(n-1) |
* \                a(n)  b(n) / \ x(n)/ \ r(n)/
*
* The diagonals A, B, and C, and the right-hand sides
* of the equations R, are provided as input to the
* subroutine, and the solution X is returned.
*
      Integer n
      Double Precision A(n), B(n), C(n), X(n), R(n)
      Double Precision BETA(100), RHO(100)
      If(n .gt. 100) STOP ' Arrays too large for TRISOLVE'
      If (b(1) .eq. 0)
+          STOP 'Zero diagonal element in TRISOLVE'
      beta(1) = b(1)
      rho(1) = r(1)
      DO j=2,n
          beta(j) = b(j) - a(j) * c (j-1) / beta(j-1)
          rho(j)  = r(j) - a(j)* rho(j-1) / beta(j-1)
          if(beta(j) .eq. 0)
+              STOP 'Zero diagonal element in TRISOLVE'
      END DO
*
* Now, for the back substitution...
*
      x(n) = rho(n) / beta(n)
      DO j = 1, n-1
          x(n-j) = ( rho(n-j)-c(n-j)*x(n-j+1) )/beta(n-j)
      END DO
      end

```

Note that the arrays A, B, C, X, and R are dimensioned n in the calling routine. BETA and RHO are used only in this subroutine, and not in the calling

routine, and so must be explicitly dimensioned here and a check performed to verify that they are sufficiently large. Since BETA will eventually be used in the denominator of an expression, it should be verified that it is nonzero.

There is one further modification that we should make before we use this code. While “efficiency” is not of overwhelming importance to us, neither should we be carelessly inefficient. Do you see where we have erred? As it stands, an entire array is being wasted — there is no need for both X and RHO ! In the elimination phase, the X array is never used, while in the back substitution phase, it is equated to the elements of RHO . Defining two distinct variables was necessary when we were developing the method of solution, and it was entirely appropriate that we used them. At that point, we didn’t know what the relationship between ρ_j and x_j was going to be. But *as a result* of our analysis we know, and we should use our understanding of that relationship as best we can. With respect to the computer code, one of the arrays is unnecessary — you should remove all references to RHO in the subroutine, in favor of the array X .

EXERCISE 3.5

After suitably modifying `TriSolve`, use it to solve the set of equations

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}. \quad (3.53)$$

You can check your result by verifying that your solution satisfies the original equations.

Cubic Spline Interpolation

Now that we see what’s involved, the solution to the cubic spline problem is fairly clear, at least in principle. From the given table of x_j ’s and $f(x_j)$ ’s, the tridiagonal set of equations of Equation (3.38) (or Equation (3.39), for the natural spline) is determined. Those equations are then solved, by a subroutine such as `TriSolve`, for the p_j'' . Knowing these, all other quantities can be determined. In pseudocode, the subroutine might look something like this:

```
Subroutine SplineInit(x,f,fp1,fpN,second,n)
```

*

```

* This subroutine is called with the first derivatives
* at x=x(1), FP1, and at x=x(n), FPN, specified, as
* well as the X's and the F's. (And n.)
*
* It returns the second derivatives in SECOND.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine.
*
* < Initialize A, B, C --- the subdiagonal, main diagonal,
*   and super-diagonal.>
* < Initialize R --- the right-hand side.>
*
* < Call TRISOLVE to solve for SECOND, the second
*   derivatives. This requires the additional array BETA.>
*
      end

```

This subroutine will certainly work. But... Are all those arrays *really* necessary? If we leave the solution of the tridiagonal system to a separate subroutine, they probably are. However, it would be just as useful to have a specialized version of TRISOLVE within the spline code. This would make the spline subroutine self-contained, which can be a valuable characteristic in itself. As we think about this, we realize that TRISOLVE will be the *major* portion of the spline routine — what we *really* need to do is to *start* with TRISOLVE, and add the particular features of spline interpolation to it. This new point of view is considerably different from what we began with, and not obvious from the beginning. The process exemplifies an important component of good programming and successful computational physics — don't get "locked in" to one way of doing things; always look to see if there is a better way. Sometimes there isn't. And sometimes, a new point of view can lead to dramatic progress.

So, let's start with TRISOLVE. To begin, we should immediately change the name of the array X to SECOND, to avoid confusion with the coordinates that we'll want to use. Then we should realize that we don't need both A and C, since the particular tridiagonal matrix used in the cubic spline problem is symmetric. In fact, we recognize that it's not necessary to have arrays for the diagonals of the matrix at all — we can evaluate them as we go! That is, instead of having a DO loop to evaluate the components of A, for example, we can evaluate the specific element we need within the Gauss elimination loop. The arrays A, B, C, and R are not needed! The revised code looks like this:

```
Subroutine SplineInit(x,f,fp1,fpn,second,n)
```

```

*
* This subroutine performs the initialization of second
* derivatives necessary for CUBIC SPLINE interpolation.
* The arrays X and F contain N values of function and the
* position at which it was evaluated, and FP1 and FPn
* are the first derivatives at  $x = x(1)$  and  $x = x(n)$ .
*
* The subroutine returns the second derivatives in SECOND.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine --- BETA is dimensioned locally.
*
      Integer n
      Double Precision X(n), F(n), Second(n)
      Double Precision FP1,FPn
      Double Precision BETA(100)
*
* In a cubic spline, the approximation to the function and
* its first two derivatives are required to be continuous.
* The primary function of this subroutine is to solve the
* set of tridiagonal linear equations resulting from this
* requirement for the (unknown) second derivatives and
* knowledge of the first derivatives at the ends of the
* interpolating region. The equations are solved by
* Gaussian elimination, restricted to tridiagonal systems,
* with A, B, and C being sub, main, and super diagonals,
* and R the right hand side of the equations.
*
      If(n .gt. 100)
+         STOP 'Array too large for SPLINE INIT'
      b1 = 2.d0*(x(2)-x(1))
      beta(1) = b1
      If (beta(1) .eq. 0)
+         STOP 'Zero diagonal element in SPLINE INIT'
      r1 = 6.d0*( (f(2)-f(1))/(x(2)-x(1)) - FP1 )
      second(1) = r1
      DO j=2,n
          IF (j.eq.n) THEN
              bj= 2.d0 * (x(n)-x(n-1))
              rj= -6.d0*((f(n)-f(n-1))/(x(n)-x(n-1))-FPn)
*
*           For j=2,...,n-1, do the following
          ELSE
              bj=2.d0*( x(j+1) - x(j-1) )

```

```

                rj=6.d0*( (f(j+1)-f( j ))/(x(j+1)-x( j ))
                        -(f( j )-f(j-1))/(x( j )-x(j-1)))
        ENDIF
*
* Evaluate the off-diagonal elements. Since the
* matrix is symmetric, A and C are equivalent.
*
        aj = x( j ) - x(j-1)
        c  = aj
        beta(j) = bj - aj * c / beta(j-1)
        second(j) = rj - aj* second(j-1) / beta(j-1)
        IF(beta(j) .eq. 0)
+           STOP 'Zero diagonal element in SPLINE INIT'
        END DO
*
* Now, for the back substitution...
*
        second(n) = second(n) / beta(n)
        DO j = 1, n-1
            c = x(n-j+1)-x(n-j)
            second(n-j) =
+           ( second(n-j) - c * second(n-j+1) )/beta(n-j)
        END DO
        end

```

This code could be made more “compact” — that is, we don’t really need to use intermediate variables like *aj* and *c*, and they could be eliminated — but the clarity might be diminished in the process. It’s far better to have a clear, reliable code than one that is marginally more “efficient” but is difficult to comprehend.

Of course, we haven’t interpolated anything yet! *SplineInit* yields the second derivatives, which we need for interpolation, but doesn’t do the interpolation itself. *SplineInit* needs to be called, once, before any interpolation is done.

The interpolation itself, the embodiment of Equation (3.31), is done in the function *Spline*:

```

        Double Precision Function Spline(xvalue,x,f,second)
*
* This subroutine performs the CUBIC SPLINE interpolation,
* after SECOND has been initialized by SPLINE INIT.
* The arrays F, SECOND, and X contain N values of function,

```

```

* its second derivative, and the positions at which they
* were evaluated.
*
* The subroutine returns the cubic spline approximation
* to the function at XVALUE.
*
* The arrays X, F, and SECOND are dimensioned in the
* calling routine.
*
      Integer n
      Double Precision Xvalue, X(n), F(n), Second(n)
*
* Verify that XVALUE is between x(1) and x(n).
*
      IF (xvalue .lt. x(1))
+       Stop 'XVALUE is less than X(1) in SPLINE'
      IF (xvalue .gt. x(n))
+       Stop 'XVALUE is greater than X(n) in SPLINE'
*
* Determine the interval containing XVALUE.
*
      j = 0
100    j = j + 1
      IF ( xvalue .gt. x(j+1) ) goto 100
*
* Xvalue is between x(j) and x(j+1).
*
* Now, for the interpolation...
*
      ...
      spline = ...
      end

```

Some of the code has been left for you to fill in.

EXERCISE 3.6

Use the cubic spline approximation to the Bessel function to find the location of its first zero.

In this spline approximation, we need to know the derivatives of the function at the endpoints. But how much do they matter? If we had simply “guessed” at the derivatives, how much would it affect the approximation? One of the advantages of a computational approach is that we answer these

questions almost as easily as we can ask them.

EXERCISE 3.7

Investigate the influence of the derivatives on the approximation to J_1 , by entering false values for FP1. Instead of the correct derivative, use $J_1'(0) = -3.0, -2.5, -2.0, \dots, 2.5$, and 3, and plot your bogus approximations against the valid one. How much of a difference does the derivative make?

Earlier, we noted that in addition to the “clamped” spline approximation in which the derivatives at the endpoints are specified, there is the “natural” spline for which derivative information is not required.

EXERCISE 3.8

Change `SplineInit` into `NaturalSplineInit`, a subroutine which initializes the natural spline. Plot the “clamped” and natural spline approximations to J_1 , and compare them. (Are changes needed in the function `Spline`?)

Occasionally, we find the derivative capability of the spline approximation to be particularly valuable. Based upon Equation (3.32), it’s easy to develop a function analogous to `Spline` that evaluates the derivative.

EXERCISE 3.9

Develop the function `SplinePrime`, which evaluates the derivative of the function, and consider the intensity of light passing through a circular aperture. We previously investigated the location of the dark fringe — past that fringe, the intensity builds towards a maximum — the location of the first light fringe. Where is it, and how bright is it? (Note that you need to approximate I/I_0 , not J_1 . Extrema of the intensity correspond to zeros of the derivative of the intensity.)

Approximation of Derivatives

If a function, and all its derivatives, are known at some point, then Taylor’s series would seem to be an excellent way to approximate the function. As a practical matter, however, the method leaves quite a lot to be desired — how often are a function and all its derivatives known? This situation is somewhat complementary to that of Lagrange interpolation — instead of know-

ing the function at many points, and having “global” information about the function, the Taylor series approximation relies upon having unlimited information about the function “local” to the point of interest. In practice, such information is usually not available. However, Taylor’s series plays a crucial role in the numerical calculation of derivatives.

There are several ways to discuss numerical differentiation. For example, we could simply take the definition of the derivative from differential calculus,

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \quad (3.54)$$

and suggest that an adequate approximation is

$$f'_h(x) = \frac{f(x+h) - f(x)}{h}. \quad (3.55)$$

Certainly, $f'_h(x)$ approaches $f'(x)$ in the limit as $h \rightarrow 0$, but how does it compare if h is other than zero? What we need is a method that will yield good approximations of *known quality* using *finite* differences; that is, the approximation should not require that the limit to zero be taken, and an estimate of the accuracy of the approximation should be provided. It’s in situations like these that the power and usefulness of the Taylor series is most obvious. Making a Taylor series expansion of $f(x+h)$ about the point $f(x)$,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \cdots, \quad (3.56)$$

we can solve for $f'(x)$ to find

$$f'(x) = \frac{1}{h} \left[f(x+h) - f(x) - \frac{h^2}{2!}f''(x) + \cdots \right]. \quad (3.57)$$

This is not an approximation — it is an expression for $f'(x)$ that has exactly the same validity as the original Taylor series from which it was derived. And the terms that appear in the expression involve the actual function of interest, $f(x)$, not some approximation to it (quadratic or otherwise). On the other hand, Equation (3.57) certainly *suggests* the approximation,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \quad (3.58)$$

By referring to Equation (3.57) we see that the error incurred in using this approximation is

$$E(h) = \frac{h}{2}f''(x) + \cdots, \quad (3.59)$$

where we've written only the leading term of the error expression — the next term will contain h^2 and for small h will be correspondingly smaller than the term exhibited. Another way of indicating the specific approximation being made is to write

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \quad (3.60)$$

where $\mathcal{O}(h)$ indicates the *order* of the approximation being made by neglecting all other terms in the total expansion.

Equation (3.60) is a *forward difference* approximation to the derivative. We can derive a different expression for $f'(x)$, starting with a Taylor series expansion of $f(x-h)$. This leads to an expression analogous to Equation (3.57),

$$f(x) = \frac{1}{h} \left[f(x) - f(x-h) + \frac{h^2}{2!} f''(x) + \cdots \right], \quad (3.61)$$

and the *backward difference* approximation,

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h). \quad (3.62)$$

Since the forward and backward difference formulas are obtained in similar ways, it's not surprising that their error terms are of the same order. Thus neither has a particular advantage over the other, in general. (An exception occurs at the limits of an interval, however. If the function is only known on the interval $a < x < b$, then the backward difference formula can not be used at $x = a$. Likewise, the forward difference formula can't be used at $x = b$.) However, the two formulas have an interesting relation to one another. In particular, the leading term in the error in one formula is of *opposite sign* compared to the other. If we *add* the two expressions, the leading term will *cancel!* Let's return to the Taylor series expansions, and include some higher order terms,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(x) + \frac{h^4}{4!} f^{iv}(x) + \cdots \quad (3.63)$$

and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!} f''(x) - \frac{h^3}{3!} f'''(x) + \frac{h^4}{4!} f^{iv}(x) + \cdots \quad (3.64)$$

Wanting an expression for $f'(x)$, we subtract these expressions to find the *central difference* formula for the derivative,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2), \quad (3.65)$$

in which the error term is

$$E(h) = -\frac{h^2}{3!}f'''(x) - \frac{h^4}{5!}f^{(5)}(x) + \cdots. \quad (3.66)$$

Not only has the error been reduced to $\mathcal{O}(h^2)$, but all the terms involving odd powers of h have been eliminated.

As a general rule, symmetric expressions are more accurate than nonsymmetric ones.

EXERCISE 3.10

Consider the function $f(x) = xe^x$. Obtain approximations to $f'(2)$, with $h = 0.5, 0.45, \dots, 0.05$, using the forward, backward, and central difference formulas. To visualize how the approximation improves with decreasing step size, plot the error as a function of h .

In the present instance, we know the correct result and so can also plot the error, as the difference between the calculated and the true value of the derivative. For any function, such as this error, that is written as

$$E(h) \approx h^n, \quad (3.67)$$

the natural logarithm is simply

$$\ln E(h) \approx n \ln h. \quad (3.68)$$

Thus, if $\ln E(h)$ is plotted versus $\ln h$, the value of n is just the slope of the line!

EXERCISE 3.11

Verify the order of the error in the backward, forward, and central difference formulas by plotting the natural log of the error versus the natural log of h in your approximation of the derivative of $f = xe^x$ at $x = 2$.

These methods can be used to evaluate higher derivatives. For example, Equations (3.63) and (3.64) can be added to eliminate $f'(x)$ from the

expression, yielding

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}(h^2) \quad (3.69)$$

with an error of

$$E(h) = -\frac{h^2}{12}f^{iv}(x) - \frac{h^4}{360}f^{vi}(x) + \dots \quad (3.70)$$

Again, we note that terms with odd powers of h are absent, a consequence of the symmetry of the expression. To develop a forward difference formula, we need to know that

$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2}{2!}f''(x) + \frac{8h^3}{3!}f'''(x) + \frac{16h^4}{4!}f^{iv}(x) + \dots \quad (3.71)$$

Combining this with Equation (3.63) yields

$$f''(x) = \frac{f(x) - 2f(x+h) + f(x+2h)}{h^2} + \mathcal{O}(h). \quad (3.72)$$

While this expression uses three function evaluations, just as the central difference formula, Equation (3.69), it's not nearly as accurate — in this approximation, the error is

$$E(h) = -hf'''(x) - \frac{7h^2}{12}f^{iv}(x) + \dots \quad (3.73)$$

The reason, of course, is that with the central difference formula the function is evaluated at points surrounding the point of interest, while with the forward difference they're all off to one side. This is entirely consistent with our previous observation concerning the interpolation of functions.

EXERCISE 3.12

Consider the function $f(x) = xe^x$. Obtain approximations to $f''(2)$, with $h = 0.5, 0.45, \dots, 0.05$, using the forward and central difference formulas. Verify the order of the error by plotting the log error versus $\log h$.

Richardson Extrapolation

We have seen how *two different* expressions can be combined to eliminate the leading error term and thus yield a more accurate expression. It is also possible to use a *single* expression to achieve the same goal. This general technique is due to L. F. Richardson, a meteorologist who pioneered numerical weather prediction in the 1920s. (His *Weather Prediction by Numerical Process*, written in 1922, is a classic in the field.) Although we will demonstrate the method with regards to numerical differentiation, the general technique is applicable, and very useful, whenever the order of the error is known.

Let's start with the central difference formula, and imagine that we have obtained the usual approximation for the derivative,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(x) + \dots \quad (3.74)$$

Using a *different* step size we can obtain a second approximation to the derivative. Then using these two expressions, we can eliminate the leading term of the error. In practice, the second expression is usually obtained by using an h twice as large as the first, so that

$$f'(x) = \frac{f(x+2h) - f(x-2h)}{4h} - \frac{4h^2}{6} f'''(x) + \dots \quad (3.75)$$

While the step size is twice as large, the error is four times as large. Dividing this expression by 4 and subtracting it from the previous one eliminates the error! Well, actually only the leading term of the error is eliminated, but it still sounds great! Solving for f' , we obtain

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \mathcal{O}(h^4), \quad (3.76)$$

a 5-point central difference formula with error given by

$$E(h) = \frac{h^4}{30} f^{(5)}(x) + \dots \quad (3.77)$$

(Yes, we know there are only 4 points used in this expression. The coefficient of the fifth point, $f(x)$, happens to be zero.)

Of course, we can do the same thing with other derivatives: using the 3-point expression of Equation (3.69), we can easily derive the 5-point

expression

$$f''(x) = \frac{-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)}{12h^2} + \mathcal{O}(h^4), \quad (3.78)$$

with

$$E(h) = \frac{h^4}{90} f^{(4)}(x) + \dots \quad (3.79)$$

Now, there are two different ways that Richardson extrapolation can be used. The first is to obtain “new” expressions, as we’ve just done, and to use these expressions directly. Be forewarned, however, that these expressions can become rather cumbersome.

Richardson extrapolation can also be used in an indirect computational scheme. We’ll carry out exactly the same steps as before, but we’ll perform them *numerically* rather than *symbolically*. Let $D_1(h)$ be the approximation to the derivative obtained from the 3-point central difference formula with step size h , and imagine that both $D_1(2h)$ and $D_1(h)$ have been calculated. Clearly, the second approximation will be better than the first. But since the order of the error in this approximation is known, we can do even better. Since the error goes as the square of the step size, $D_1(2h)$ must contain four times the error contained in $D_1(h)$! The difference between these two approximations is then three times the error of the second. But the difference is something we can easily calculate, so in fact we can calculate the error! (Or at least, its leading term.) The “correct” answer, $D_2(h)$, is then obtained by simply subtracting this calculated error from the second approximation,

$$\begin{aligned} D_2(h) &= D_1(h) - \left[\frac{D_1(2h) - D_1(h)}{2^2 - 1} \right] \\ &= \frac{4D_1(h) - D_1(2h)}{3}. \end{aligned} \quad (3.80)$$

Of course, $D_2(h)$ is not the *exact* answer, since we’ve only accounted for the leading term in the error. Since the central difference formulas have error terms involving only even powers of h , the error in $D_2(h)$ must be $\mathcal{O}(h^4)$. Thus $D_2(2h)$ contains 2^4 times as much error as $D_2(h)$, and so this error can be removed to yield an even better estimate of the derivative,

$$\begin{aligned} D_3(h) &= D_2(h) - \left[\frac{D_2(2h) - D_2(h)}{2^4 - 1} \right] \\ &= \frac{16D_2(h) - D_2(2h)}{15}. \end{aligned} \quad (3.81)$$

This processes can be continued indefinitely, with each improved estimated given by

$$\begin{aligned} D_{i+1}(h) &= D_i(h) - \left[\frac{D_i(2h) - D_i(h)}{2^{2i} - 1} \right] \\ &= \frac{2^{2i} D_i(h) - D_i(2h)}{2^{2i} - 1}. \end{aligned} \quad (3.82)$$

To see how this works, consider the function $f(x) = x e^x$ and calculate $D_1(h)$ at $x = 2$ for different values of the step size h . It's convenient to arrange these in a column, with each succeeding step size half the previous one. The entries in this column are then used to evaluate $D_2(h)$, which can be listed in an adjoining column. In turn, these entries are combined to yield D_3 , and so on. We thus build a table of extrapolations, with the most accurate approximation being the bottom rightmost entry:

$f'(2)$				
h	D_1	D_2	D_3	D_4
0.4	23.16346 42931			
0.2	22.41416 06570	22.16439 27783		
0.1	22.22878 68803	22.16699 56214	22.16716 91443	
0.05	22.18256 48578	22.16715 75170	22.16716 83100	22.16716 82968

While we've suggested that this table was created a column at a time, that's not the way it would be done in practice, of course. Rather, it would be built a *row* at a time, as the necessary constituents of the various approximations became available. For example, the first entry in the table is $D_1(0.4)$ and the second entry is $D_1(0.2)$. This is enough information to evaluate $D_2(0.2)$, which fills out that row. (Note that this entry is already more accurate than the last 3-point central difference approximation in the table, $D_1(0.05)$.) If we wanted to obtain the derivative to a specific accuracy, we would compare this last approximation, $D_2(0.2)$, to the best previous approximation, $D_1(0.2)$. In this case, we only have 2-digit accuracy, so we would halve h and evaluate $D_1(0.1)$. That would enable $D_2(0.1)$ to be evaluated, which would then be used to evaluate $D_3(0.1)$, completing that row. We now have 4-significant-digit agreement between $D_2(0.1)$ and $D_3(0.1)$. Halving h one more time, the fourth row of the table can be generated. The last entries, $D_3(0.05)$ and $D_4(0.05)$, agree to 9 significant digits; $D_4(0.05)$ is actually accurate to all 12 digits displayed.

The coding of this extrapolation scheme can initially be a little intimidating, although (as we shall see) it isn't really all that difficult. As often happens, we will start with a general outline and build upon it — there's no

reason to believe that anyone ever writes a complicated computer code off the top of their head. In fact, quite the opposite is true: the more complicated the code, the more methodical its development should be. From reflection upon the extrapolation table we created, we realize that the “next” row is generated with one newly evaluated derivative and the entries of the “previous” row. So, we’ll have an “old” row, and a “new” one, each expressed as an array. An outline of the code might look something like this:

```

*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This code fragment illustrates the RICHARDSON
* EXTRAPOLATION scheme for the calculation of a derivative.
*
* 5 columns will be stored in each of the OLD and NEW
* rows of the extrapolation table. H is the step size,
* which is halved with each new row.
*
      Double Precision OLD(5),NEW(5),x,h,DF
      Integer iteration, i
      ...
      X = < point where derivative is to be evaluated >
      h = < initial step size >
      iteration = 0 <keep track of number of iterations>
100  iteration = iteration + 1
      DO i = 1, 5
          old(i) = new(i)
      END DO
*      <evaluate the derivative DF using the 3-pt formula>
      new(1) = DF
*
* The following IF-statement structure calculates the
* appropriate extrapolations.
*
      IF (iteration .ge. 2) new(2)=( 4*new(1) - old(1) )/ 3
      IF (iteration .ge. 3) new(3)=(16*new(2) - old(2) )/15
      IF (iteration ....
*
      h = h/2.DO
      GOTO 100
      ...

```

This is a start. We’ve dimensioned the arrays to be 5, although we may need to change this later. We’ve established the basic loop in which the derivative

will be evaluated and extrapolations are made, and we've redefined the step size at the *end* of the loop. However, we haven't provided any way for the program to STOP! The first thing to do to this code, then, is to provide a way to terminate execution once a specified tolerance for the accuracy of the derivative has been met. You will also want to provide a graceful exit if the tolerance isn't met, after a "reasonable" number of iterations. You might think that more entries should be allowed in the rows — however, experience suggests that the extrapolations don't improve indefinitely, so that the fourth or fifth column is about as far as one really wants to extrapolate. By the way, this general extrapolation procedure can be used any time that the order of the error is known — later, we will use Richardson extrapolation to calculate integrals. In the general case, however, odd powers of h will be present in the error terms and so the coefficients used here would not be appropriate.

■ EXERCISE 3.13

Modify the code fragment to calculate the *second derivative* of $f(x) = xe^x$, using the 3-point expression of Equation (3.69) and Richardson extrapolation, and generate a table of extrapolates in the same way that the first derivative was evaluated earlier.

Curve Fitting by Least Squares

To this point, we have discussed *interpolation* in which the approximating function passes through the given points. But what if there is some "scatter" to the points, such as will be the case with actual experimental data? A standard problem is to find the "best" fit to the data, without requiring that the approximation actually coincides with it any of the specific data points. As an example, consider an experiment in which a ball is dropped, and its velocity is recorded as a function of time. From this data, we might be able to obtain the acceleration due to gravity. (See Figure 3.3.)

Of course, we have to define what we mean by "best fit." We'll take a very simple definition, requiring a certain estimate of the error to be a minimum. Let's make a *guess* at the functional form of the data — in our example, the guess will be

$$v(t) = a + bt \tag{3.83}$$

— and evaluate the *difference* between this guess, evaluated at t_i , and the velocity measured at $t = t_i$, v_i . Because the difference can be positive or negative, we'll square this difference to obtain a non-negative number. Then, we'll

add this to the estimate from all other points. We thus use

$$S = \sum_{i=1}^N (v(t_i) - v_i)^2 \quad (3.84)$$

as an estimate of the error. Our approximation will be obtained by making this error as small as possible — hence the terminology “least squares fit.”

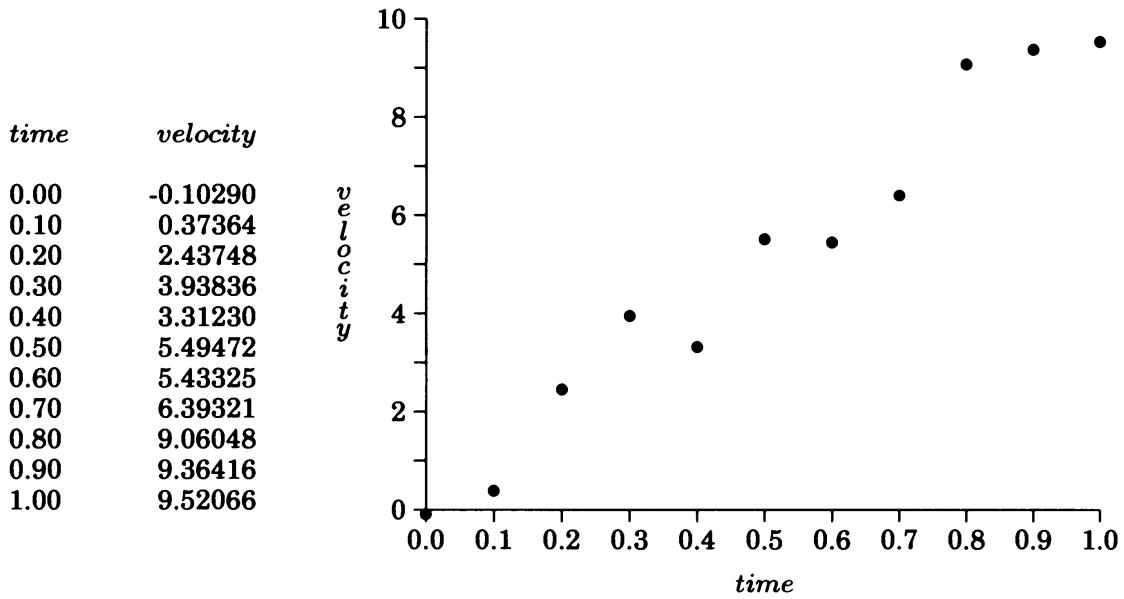


FIGURE 3.3 The measured velocity of a particle at various times.

In general, $v(t)$ will be written in terms of unknown parameters, such as the a and b in Equation (3.83). Let’s think of varying a , for example, so as to minimize S — a minimum will occur whenever $\partial S/\partial a$ is zero and the second derivative is positive. That is, the error will be an extremum when

$$\frac{\partial S}{\partial a} = \sum_{i=1}^N 2(v(t_i) - v_i) \frac{\partial v}{\partial a} \Big|_{t=t_i} = \sum_{i=1}^N 2(a + bt_i - v_i) = 0. \quad (3.85)$$

And since

$$\frac{\partial^2 S}{\partial a^2} = \sum_{i=1}^N 2 > 0, \quad (3.86)$$

we will have found a minimum. Of course, we'll have similar equations for b ,

$$\frac{\partial S}{\partial b} = \sum_{i=1}^N 2(v(t_i) - v_i) \frac{\partial v}{\partial b} \Big|_{t=t_i} = \sum_{i=1}^N 2(a + bt_i - v_i)t_i = 0 \quad (3.87)$$

and

$$\frac{\partial^2 S}{\partial a^2} = \sum_{i=1}^N 2t_i^2 > 0, \quad (3.88)$$

so that the error is minimized with respect to b as well. We thus arrive at the equations

$$aN + b \sum_{i=1}^N t_i = \sum_{i=1}^N v_i, \quad (3.89)$$

and

$$a \sum_{i=1}^N t_i + b \sum_{i=1}^N t_i^2 = \sum_{i=1}^N v_i t_i. \quad (3.90)$$

If a and b can be found such that these equations hold, then we will have found a minimum in the error.

EXERCISE 3.14

Solve for a and b . Plot the velocity data and your “best fit” to that data.

In a more complicated situation, the equations might be more difficult to solve. That is, most of us have little difficulty with solving two equations for two unknowns. But 5 equations, or 25 equations, is a different matter. Before we go on, we really need to discuss the solution to an arbitrary linear set of simultaneous solutions, a standard topic of linear algebra.

Gaussian Elimination

Consider the set of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \quad (3.91)$$

We'll solve this set via Gauss elimination. In discussing tridiagonal systems, we hit upon the main idea of the process: combine two of the equations in such a way as to eliminate one of the variables, and replace one of the original equations by the reduced one. Although the system of equations confronting us now is not as simple as the tridiagonal one, the process is exactly the same.

We'll start by eliminating x_1 in all but the first equation. Consider the i -th equation,

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i. \quad (3.92)$$

Multiplying the first equation by a_{i1}/a_{11} and subtracting it from the i -th equation gives us

$$\begin{aligned} (a_{i2} - \frac{a_{i1}}{a_{11}}a_{12})x_2 + (a_{i3} - \frac{a_{i1}}{a_{11}}a_{13})x_3 + \cdots + (a_{in} - \frac{a_{i1}}{a_{11}}a_{1n})x_n \\ = b_i - \frac{a_{i1}}{a_{11}}b_1, \quad i = 2, \dots, N. \end{aligned} \quad (3.93)$$

After x_1 has been eliminated in equations 2 through N , we repeat the process to eliminate x_2 from equations 3 through N . Eventually, we arrive at an upper triangular set of equations that is easily solved by back substitution.

Now, let's repeat some of what we just said, but in matrix notation. We start with the set of Equations (3.91), written as a matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}. \quad (3.94)$$

To eliminate x_1 , we combined equations in a particular way. That process can be expressed as a matrix multiplication — starting with

$$\mathbf{Ax} = \mathbf{b}, \quad (3.95)$$

we multiplied by the matrix \mathbf{M}_1 , where

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -a_{21}/a_{11} & 1 & 0 & \cdots & 0 \\ -a_{31}/a_{11} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_{n1}/a_{11} & 0 & 0 & \cdots & 1 \end{bmatrix}, \quad (3.96)$$

yielding the equation $\mathbf{M}_1 \mathbf{A} \mathbf{x} = \mathbf{M}_1 \mathbf{b}$, where

$$\mathbf{M}_1 \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} - \frac{a_{21}}{a_{11}} a_{12} & a_{23} - \frac{a_{21}}{a_{11}} a_{13} & \dots & a_{2n} - \frac{a_{21}}{a_{11}} a_{1n} \\ 0 & a_{32} - \frac{a_{31}}{a_{11}} a_{12} & a_{33} - \frac{a_{31}}{a_{11}} a_{13} & \dots & a_{3n} - \frac{a_{31}}{a_{11}} a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} - \frac{a_{n1}}{a_{11}} a_{12} & a_{n3} - \frac{a_{n1}}{a_{11}} a_{13} & \dots & a_{nn} - \frac{a_{n1}}{a_{11}} a_{1n} \end{bmatrix} \quad (3.97)$$

and

$$\mathbf{M}_1 \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 - \frac{a_{21}}{a_{11}} b_1 \\ b_3 - \frac{a_{31}}{a_{11}} b_1 \\ \vdots \\ b_n - \frac{a_{n1}}{a_{11}} b_1 \end{bmatrix}, \quad (3.98)$$

which is just what we would expect from Equation (3.93). The next step, to eliminate x_2 from equations 3 through N , is accomplished by multiplying by \mathbf{M}_2 , and so on.

A particularly useful variation of this method is due to Crout. In it, we write \mathbf{A} as a product of a lower triangular matrix \mathbf{L} and an upper triangular matrix \mathbf{U} ,

$$\mathbf{A} = \mathbf{L} \mathbf{U}, \quad (3.99)$$

or

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & 1 & u_{23} & \dots & u_{2n} \\ 0 & 0 & 1 & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}. \quad (3.100)$$

Written out like this, the meaning of “lower” and “upper” triangular matrix is pretty clear. It might seem that finding \mathbf{L} and \mathbf{U} would be difficult, but

actually it's not. Consider an element in the first column of A , and compare it to the corresponding element of the matrix product on the right side. We find that $a_{11} = l_{11}$, $a_{21} = l_{21}$, $a_{31} = l_{31}$, and so on, giving us the first column of L . From the first row of A we find

$$\begin{aligned} a_{12} &= l_{11}u_{12} \\ a_{13} &= l_{11}u_{13} \\ &\vdots \\ a_{1n} &= l_{11}u_{1n}. \end{aligned} \quad (3.101)$$

But we already know l_{11} , and so we can solve these equations for the first row of U ,

$$u_{1j} = a_{1j}/l_{11}, \quad j = 2, \dots, N. \quad (3.102)$$

We now move to the second column of A , which gives us

$$\begin{aligned} a_{22} &= l_{21}u_{12} + l_{22} \\ a_{32} &= l_{31}u_{12} + l_{32} \\ &\vdots \\ a_{n2} &= l_{n1}u_{12} + l_{n2}. \end{aligned} \quad (3.103)$$

Again, we know u_{12} and all the l_{i1} appearing in these equations, so we can solve them for the second column of L ,

$$l_{i2} = a_{i2} - l_{i1}u_{12}, \quad i = 2, \dots, N. \quad (3.104)$$

We then consider the second row of A , and so on. In general, from the k -th column of A we find that

$$l_{ik} = a_{ik} - \sum_{j=1}^{k-1} l_{ij}u_{jk}, \quad i = k, k+1, \dots, n, \quad (3.105)$$

while from the k -th row of A we find that

$$u_{kj} = \frac{a_{kj} - \sum_{i=1}^{k-1} l_{ki}u_{ij}}{l_{kk}}, \quad j = k+1, k+2, \dots, n. \quad (3.106)$$

By alternating between the columns and rows, we can solve for all the elements of L and U .

Recall that the original problem is to find \mathbf{x} in the matrix equation

$$\mathbf{Ax} = \mathbf{b}, \quad (3.107)$$

or, equivalently, in

$$\mathbf{LUx} = \mathbf{b}, \quad (3.108)$$

where we've replaced \mathbf{A} by \mathbf{LU} . Defining $\mathbf{z} = \mathbf{Ux}$, this is the same as

$$\mathbf{Lz} = \mathbf{b}, \quad (3.109)$$

an equation for the unknown \mathbf{z} . But this is “easy” to solve! In terms of the component equations, we have

$$\begin{array}{ccccccc} l_{11}z_1 & & & & & = & b_1 \\ l_{21}z_1 & +l_{22}z_2 & & & & = & b_2 \\ l_{31}z_1 & +l_{32}z_2 & +l_{33}z_3 & & & = & b_3 \\ \vdots & \vdots & \vdots & \ddots & & \vdots & \vdots \\ l_{n1}z_1 & +l_{n2}z_2 & +l_{n3}z_3 & \dots & +l_{nn}z_n & = & b_n \end{array} \quad (3.110)$$

From the first of these we find $z_1 = b_1/l_{11}$, and from subsequent equations we find

$$\begin{aligned} z_i &= \frac{b_i - l_{i1}z_1 - l_{i2}z_2 \cdots - l_{i,i-1}z_{i-1}}{l_{ii}} \\ &= \frac{b_i - \sum_{k=1}^{i-1} l_{ik}z_k}{l_{ii}}, \quad i = 2, \dots, N, \end{aligned} \quad (3.111)$$

by *forward substitution*. Having found \mathbf{z} , we then solve

$$\mathbf{Ux} = \mathbf{z} \quad (3.112)$$

for \mathbf{x} . Since \mathbf{U} is upper diagonal, this equation is also easily solved — it's already in the form required for *backward substitution*. We have

$$\begin{array}{ccccccc} x_1 & +u_{12}x_2 & +u_{13}x_3 & \dots & +u_{1n}x_n & = & z_1 \\ & x_2 & +u_{23}x_3 & \dots & +u_{2n}x_n & = & z_2 \\ & & x_3 & \dots & +u_{3n}x_n & = & z_3 \\ & & & \ddots & \vdots & \vdots & \vdots \\ & & & & x_n & = & z_n \end{array}, \quad (3.113)$$

from which we find

$$x_{n-i} = z_{n-i} - \sum_{k=1}^{i-1} u_{n-i,k} x_k, \quad i = 1, n-1. \quad (3.114.)$$

In some problems we might be required to solve $Ax = b$ for many different b 's. In that situation we would perform the decomposition only once, storing L and U and using them in the substitution step to find x .

Superficially, this process looks much different from Gaussian elimination. However, as you move through it, you find that you're performing exactly the same arithmetic steps as with Gaussian elimination — actually, the Crout decomposition is nothing more than an efficient bookkeeping scheme. It has the additional advantage of not requiring b until the substitution phase. Thus the decomposition is independent of b , and once the decomposition has been accomplished, only the substitution phases need to be performed to solve for x with a different b . The LU decomposition is also efficient in terms of storage, in that the original array A can be used to store its LU equivalent. That is, the array A will initially contain the elements of A , but can be overwritten so as to store L and U in the scheme

$$[A] = \begin{bmatrix} L(1,1) & U(1,2) & U(1,3) & \dots & U(1,N) \\ L(2,1) & L(2,2) & U(2,3) & \dots & U(2,N) \\ L(3,1) & L(3,2) & L(3,3) & \dots & U(3,N) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L(N,1) & L(N,2) & L(N,3) & \dots & L(N,N) \end{bmatrix}. \quad (3.115)$$

There is a problem with the algorithm, however, as it's been presented. If any l_{qq} is zero, the computer will attempt to divide by zero. This could happen, for example, in trying to solve the set of equations

$$\begin{aligned} x_2 &= 2 \\ x_1 &= 1. \end{aligned} \quad (3.116)$$

These equations clearly have a trivial solution, but in determining the first row of U from Equation (3.102) the algorithm will attempt to evaluate u_{12} as $1/0$. The remedy is simply to interchange the order of the two equations. In general, we'll search for the largest element of the k -th column of L , and interchange the rows to move that element onto the diagonal. If the largest element is zero, then the system of equations has no unique solution! In some cases it can also happen that the coefficients vary by orders of magnitude. This makes determining the "largest" difficult. To make the comparison of the coefficients meaningful, the equations should be scaled so that the largest

coefficient is made to be unity. (We note that this scaling is done for comparison purposes only, and need not be included as a step in the actual computation.) The swapping of rows and scaling makes the coding a little involved, and so we'll provide a working subroutine, LUSolve, that incorporates these complications.

```

      Subroutine LUSolve( A, x, b, det, ndim, n )
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine solves the linear set of equations
*
*           A x = b
*
* by the method of L U decomposition.
*
* INPUT:      ndim    the size of the arrays, as dimensioned
*                in the calling routine
*              n       the actual size of the arrays for
*                this problem
*              A       an n by n array of coefficients,
*                altered on output
*              b       a vector of length n
*
* OUTPUT:     x       the 'solution' vector
*              det     the determinant of A.  If the
*                determinant is zero, A is SINGULAR.
*
*
*                               1/1/93
*
      integer ndim,n,order(100),i,j,k, imax,itemp
      double precision a(ndim,ndim),x(ndim),b(ndim),det
      double precision scale(100), max, sum, temporary
      if(n.gt.100)stop ' n too large in LUSolve'
      det = 1.d0
*
* First, determine a scaling factor for each row.  (We
* could "normalize" the equation by multiplying by this
* factor.  However, since we only want it for comparison
* purposes, we don't need to actually perform the
* multiplication.)
*
      DO i = 1, n
          order(i) = i

```

```

        max = 0.d0
        DO j = 1, n
            if( abs(a(i,j)) .gt. max) max = abs(a(i,j))
        END DO
        scale(i) = 1.d0/max
    END DO

*
* Start the LU decomposition. The original matrix A
* will be overwritten by the elements of L and U as
* they are determined. The first row and column
* are specially treated, as is L(n,n).
*
        DO k = 1, n-1
*
* Do a column of L
*
            IF( k .eq. 1 ) THEN
*
* No work is necessary.
*
            ELSE
*
* Compute elements of L from Eq. (3.105).
*
                DO i = k, n
                    !
                    sum = a(i,k)
                    DO j = 1, k-1
                        sum = sum - a(i,j)*a(j,k)
                    END DO
                    a(i,k) = sum
                    ! Put L(i,k) into A.
                END DO
            ENDIF

*
* Do we need to interchange rows? We want the largest
* (scaled) element of the recently computed column of L
* moved to the diagonal (k,k) location.
*
            max = 0.d0
            DO i = k, n
                IF(scale(i)*a(i,k) .ge. max) THEN
                    max = scale(i)*a(i,k)
                    imax=i
                ENDIF
            END DO

*
* Largest element is L(imax,k). If imax=k, the largest
* (scaled) element is already on the diagonal.

```

```

*
      IF(imax .eq. k) THEN
*         No need to exchange rows.
      ELSE
*         Exchange rows...
*
          det = -det
          DO j = 1, n
              temporary = a(imax,j)
              a(imax,j) = a(k,j)
              a(k,j)    = temporary
          END DO
*
*         scale factors...
*
          temporary = scale(imax)
          scale(imax) = scale(k)
          scale(k)    = temporary
*
*         and record changes in the ordering
*
          itemp      = order(imax)
          order(imax) = order(k)
          order(k)    = itemp
*
      ENDIF
      det = det * a(k,k)
*
* Now compute a row of U.
*
      IF(k.eq.1) THEN
*         The first row is treated special, see Eq. (3.102).
*
          DO j = 2, n
              a(1,j) = a(1,j) / a(1,1)
          END DO
      ELSE
*         Compute U(k,j) from Eq. (3.106).
*
          DO j = k+1, n
              sum = a(k,j)
              DO i = 1, k-1
                  sum = sum - a(k,i)*a(i,j)
              END DO

```

```

*   Put the element U(k,j) into A.
*
*           a(k,j) = sum / a(k,k)
*           END DO
*       ENDIF
1000    CONTINUE
*
* Now, for the last element of L
*
*       sum = a(n,n)
*       DO j = 1, n-1
*           sum = sum - a(n,j)*a(j,n)
*       END DO
*       a(n,n) = sum
*       det = det * a(n,n)
*
* LU decomposition is now complete.
*
* We now start the solution phase.  Since the equations
* have been interchanged, we interchange the elements of
* B the same way, putting the result into X.
*
*       DO i = 1, n
*           x(i) = b( order(i) )
*       END DO
*
* Forward substitution...
*
*       x(1) = x(1) / a(1,1)
*       DO i = 2, n
*           sum = x(i)
*           DO k = 1, i-1
*               sum = sum - a(i,k)*x(k)
*           END DO
*           x(i) = sum / a(i,i)
*       END DO
*
* and backward substitution...
*
*       DO i = 1, n-1
*           sum = x(n-i)
*           DO k = n-i+1, n
*               sum = sum - a(n-i,k)*x(k)
*           END DO

```

```

        x(n-i) = sum
    END DO
*
* and we're done!
*
    end

```

A calculation of the determinant of the matrix has been included. The determinant of a product of matrices is just the product of the determinants of the matrices. For triangular matrices, the determinant is just the product of the diagonal elements, as you'll find if you try to expand it according to Cramer's rule. So after the matrix A has been written as LU , the evaluation of the determinant is straightforward. Except, of course, that permuting the rows introduces an overall sign change, which must be accounted for. If the determinant is zero, no unique, nontrivial solution exists; small determinants are an indication of an ill-conditioned set of equations whose solution might be very difficult.

EXERCISE 3.15

You might want to test the subroutine by solving the equations

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \quad (3.117)$$

which you've seen before.

EXERCISE 3.16

As an example of a system that might be difficult to solve, consider

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}. \quad (3.118)$$

The matrix with elements $H_{ij} = 1/(i + j - 1)$ is called the Hilbert matrix, and is a classic example of an ill-conditioned matrix. With double precision arithmetic, you are equipped to solve this particular problem. However, as the dimension of the array increases it becomes

futile to attempt a solution. Even for this 5×5 matrix, the determinant is surprisingly small — be sure to print the determinant, as well as to find the solutions.

Before returning to the problem of curve fitting by least squares, we need to emphasize the universality of systems of linear equations, and hence the importance of Gaussian elimination in their solution. You've probably seen this problem arise many times already, and you will surely see it many more times. For example, consider a typical problem appearing in university physics courses: electrical networks, as seen in Figure 3.4.

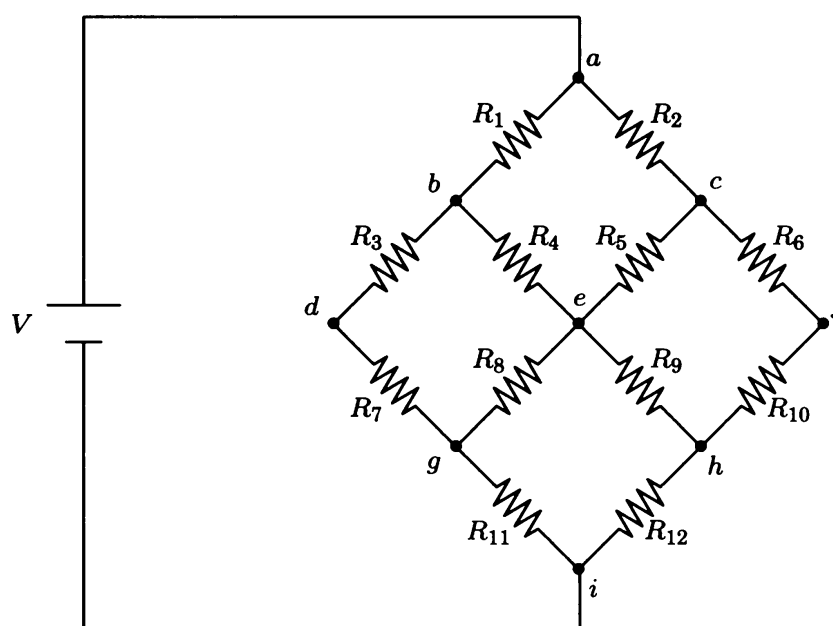


FIGURE 3.4 An electrical network requiring the solution of a set of simultaneous linear equations.

Let's arbitrarily define the potential at reference point i to be zero; the potential at a is then V . Given the resistances and the applied voltage, the problem is to determine the potential at each of the other points b, c, \dots, h . Current enters the resistance grid at a , passes through the resistors, and returns to the battery through point i . Since there is no accumulation of charge at any point, any current flowing into a node must also flow out — this is simply a statement of the conservation of electrical charge. As an example, the current flowing through R_1 to the point b must equal the current flowing through resistors R_3 and R_4 away from the point b . And from Ohm's law we know that the voltage drop across a resistor is the product of the resistance and the current, or that the current is simply the voltage drop divided by the

resistance. For point b we can then write

$$\frac{V - V_b}{R_1} = \frac{V_b - V_d}{R_3} + \frac{V_b - V_e}{R_4}, \quad (3.119)$$

and similar equations for all the other points in the network. Consider a specific network in which $R_1 = R_3 = R_4 = R_9 = R_{10} = R_{12} = 1$ ohm, $R_2 = R_5 = R_6 = R_7 = R_8 = R_{11} = 2$ ohms, and $V = 10$ volts. We easily find the following set of 7 equations describing the system,

$$\begin{array}{rcccccccl} 3V_b & & -V_d & -V_e & & & & = & 10 \\ & 3V_c & & -V_e & -V_f & & & = & 10 \\ 2V_b & & -3V_d & & & +V_g & & = & 0 \\ 2V_b & +V_c & & -6V_e & & +V_g & +2V_h & = & 0 \\ & V_c & & & -3V_f & & +2V_h & = & 0 \\ & & V_d & +V_e & & -3V_g & & = & 0 \\ & & & V_e & +V_f & & -3V_h & = & 0 \end{array} \quad (3.120)$$

EXERCISE 3.17

Use `LUsolve` to find the voltages in the described electrical network.

Much more complicated networks can also be considered. Any network involving simply batteries and resistors will lead to a set of simultaneous linear equations such as these. Incorporation of capacitors and inductors, which depend upon the time rate of change of the voltage, give rise to linear differential equations. Including transistors and diodes further complicates the problem by adding nonlinearity to the network.

General Least Squares Fitting

We can now turn our attention to least squares fitting with an arbitrary polynomial of degree m . We'll write the polynomial as

$$p(x) = c_0 + c_1x + c_2x^2 + \cdots + c_mx^m. \quad (3.121)$$

The sum of the square of the errors is then

$$S = \sum_{j=1}^N (p(x_i) - y_i)^2, \quad (3.122)$$

where $p(x)$ is our assumed functional form for the data known at the N points (x_i, y_i) . This error is to be minimized by an appropriate choice of the coefficients c_i ; in particular, we require

$$\begin{aligned}\frac{\partial S}{\partial c_0} &= \sum_{j=1}^N 2(c_0 + c_1 x_j + c_2 x_j^2 + \cdots + c_m x_j^m - y_j) = 0, \\ \frac{\partial S}{\partial c_1} &= \sum_{j=1}^N 2x_j (c_0 + c_1 x_j + c_2 x_j^2 + \cdots + c_m x_j^m - y_j) = 0, \\ &\vdots \\ \frac{\partial S}{\partial c_m} &= \sum_{j=1}^N 2x_j^m (c_0 + c_1 x_j + c_2 x_j^2 + \cdots + c_m x_j^m - y_j) = 0.\end{aligned}\quad (3.123)$$

This gives us $m + 1$ equations for the $m + 1$ unknown coefficients c_j ,

$$\begin{aligned}c_0 N &+ c_1 \sum x_j &+ c_2 \sum x_j^2 &+ \cdots &+ c_m \sum x_j^m - \sum y_j &= 0, \\ c_0 \sum x_j &+ c_1 \sum x_j^2 &+ c_2 \sum x_j^3 &+ \cdots &+ c_m \sum x_j^{m+1} - \sum x_j y_j &= 0, \\ \vdots &\vdots &\vdots &&\vdots &= 0, \\ c_0 \sum x_j^m &+ c_1 \sum x_j^{m+1} &+ c_2 \sum x_j^{m+2} &+ \cdots &+ c_m \sum x_j^{m+m} - \sum x_j^m y_j &= 0.\end{aligned}\quad (3.124)$$

These simultaneous linear equations, also known as *normal equations*, can be solved by `LUsolve` for the coefficients.

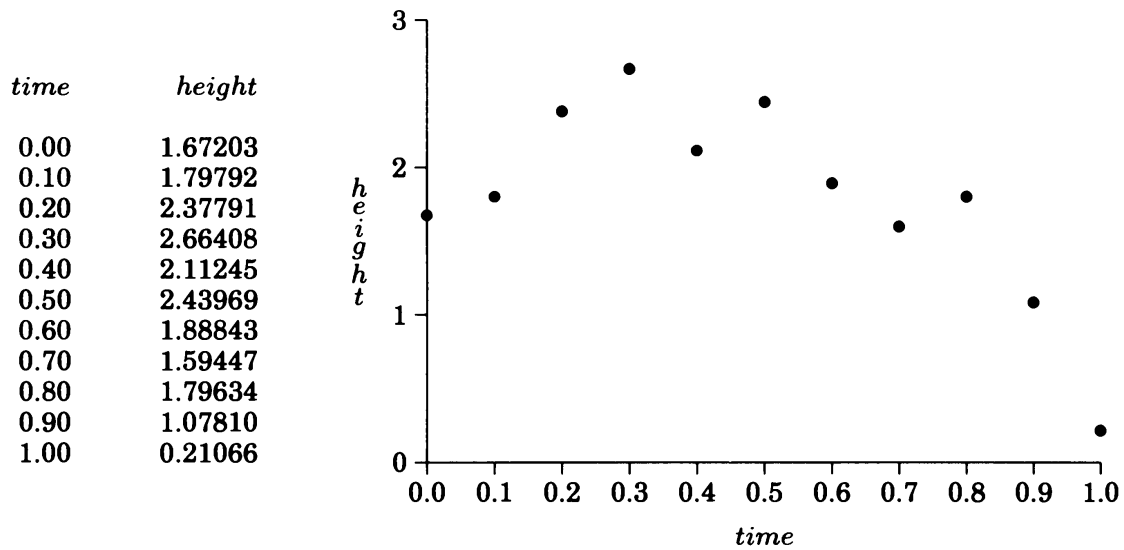


FIGURE 3.5 The measured height of a particle at various times.

In another experiment, an object has been hurled into the air and its position measured as a function of time. The data obtained in this hypothetical experiment are contained in Figure 3.5. From our studies of mechanics we know that the height should vary as the square of the time, so instead of a linear fit to the data we should use a quadratic one.

EXERCISE 3.18

Fit a quadratic to the position data, using least squares, and determine g , the acceleration due to gravity.

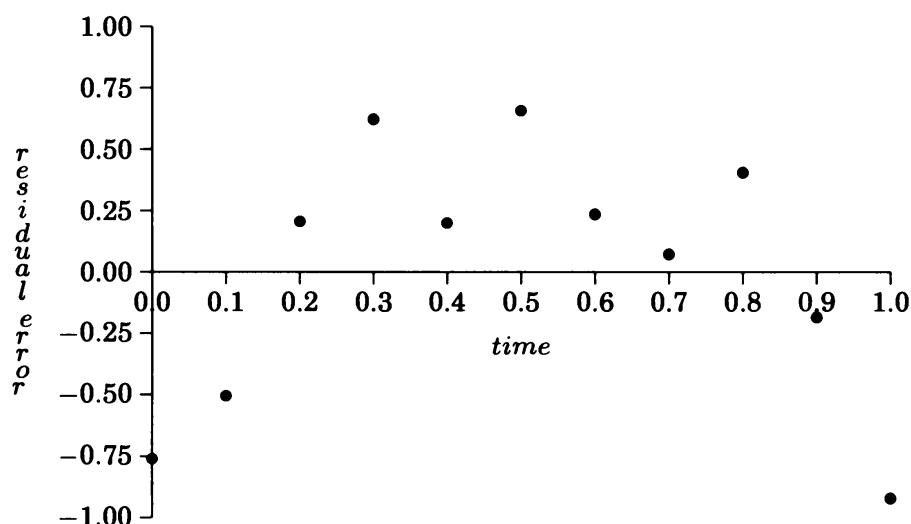


FIGURE 3.6 The residual error from the linear least squares fit to the data of Figure 3.5.

For some problems, particularly if the functional dependence of the data isn't known, there's a temptation to use a high order least squares fit. This approach didn't work earlier when we were interpolating data, and for the same reasons it won't work here either. Simply using a curve with more parameters doesn't guarantee a *better* fit. To illustrate, let's return to the position data and fit a linear, a quadratic, and a cubic polynomial to the data. For the linear fit, the least squares error is computed to be 2.84, which seems rather large given the magnitudes of our data. Even more revealing is the residual error, $y_i - p(x_i)$, which is plotted in Figure 3.6. The magnitudes of the error tells us the fit is not very good. Note that the residual error is almost systematic — at first, it's negative, and then it's positive, and then it's negative again. If we had a good fit, we would expect that the sign of the error would be virtually random. In particular, we would expect that the sign of the error

at one point would be independent of the sign at the previous point, so that we would expect the sign of the error to change about half the time. But in these residuals, the error only changes sign twice. This *strongly* suggests a functional dependence in the data that we have not included in our polynomial approximation.

But you have already determined the best quadratic fit, and found that the least squares error was about 0.52. The quadratic fit is thus a considerable improvement over the linear one. Moreover, the residual error is smaller in magnitude and more scattered than for the linear fit. In fact, there are 7 sign changes in the error, much more consistent with the picture of the approximation “fitting” through the scatter in the data than was the linear fit.

So we have found a good fit to the data, having both a small magnitude in the error and a healthy scatter in the signs of the residual errors. The prudent course of action is to quit! And what if we were to consider a higher degree polynomial? If there were no scatter in the data, then trying to fit a cubic would yield a c_3 coefficient that was zero. Of course, there’s always some scatter in the data, so this rarely happens. But the fit is no better than we already have — we find that $S = 0.52$ again. The unmistakable conclusion is that there is no advantage in going to higher order approximations.

To summarize: the primary factor in determining a good least squares fit is the validity of the functional form to which you’re fitting. Certainly, theoretical or analytic information about the physical problem should be incorporated whenever it’s available. The residual error, the difference between the determined “best fit” and the actual data, is a good indicator of the quality of the fit, and can suggest instances when a systematic functional dependence has been overlooked.

Least Squares and Orthogonal Polynomials

Although fitting data is a primary application of the method of least squares, the method can also be applied to the approximation of functions. Consider, for example, a known function $f(x)$ which we want to approximate by the polynomial $p(x)$. Instead of a Taylor series expansion, we’ll try to find the *best* polynomial approximation, in the least squares sense, to the function $f(x)$. To this end we replace the sum over data points by an integral, and define the error as being

$$S = \int_a^b (f(x) - p(x))^2 dx. \quad (3.125)$$

We'll obtain normal equations just as before, by minimizing S with respect to the coefficients in the approximating polynomial.

Consider a specific example: what quadratic polynomial best approximates $\sin \pi x$ between 0 and 1? We define the least squares error as

$$S = \int_0^1 (\sin \pi x - (c_0 + c_1 x + c_2 x^2))^2 dx, \quad (3.126)$$

and take the partial derivatives of S with respect to the coefficients to determine the normal equations

$$c_0 \int_0^1 dx + c_1 \int_0^1 x dx + c_2 \int_0^1 x^2 dx = \int_0^1 \sin \pi x dx, \quad (3.127)$$

$$c_0 \int_0^1 x dx + c_1 \int_0^1 x^2 dx + c_2 \int_0^1 x^3 dx = \int_0^1 x \sin \pi x dx, \quad (3.128)$$

and

$$c_0 \int_0^1 x^2 dx + c_1 \int_0^1 x^3 dx + c_2 \int_0^1 x^4 dx = \int_0^1 x^2 \sin \pi x dx. \quad (3.129)$$

Performing the integrals, we find

$$\begin{aligned} c_0 + \frac{1}{2}c_1 + \frac{1}{3}c_2 &= \frac{2}{\pi}, \\ \frac{1}{2}c_0 + \frac{1}{3}c_1 + \frac{1}{4}c_2 &= \frac{1}{\pi}, \\ \frac{1}{3}c_0 + \frac{1}{4}c_1 + \frac{1}{5}c_2 &= \frac{\pi^2 - 4}{\pi^3}. \end{aligned} \quad (3.130)$$

We can solve this set of equations for the coefficients, and thus determine the “best” quadratic approximation to be

$$p(x) = -0.0505 + 4.1225x - 4.1225x^2, \quad 0 < x < 1. \quad (3.131)$$

This expression certainly would not have been found by a Taylor series expansion. It is, in fact, fundamentally different. Whereas the Taylor series is an expansion about a point, the least squares fit results from consideration of the function over a specific region.

In principle, we could find higher order polynomials giving an even better approximation to the function. But Equation (3.130) seems familiar —

writing it in matrix form we have

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 2/\pi \\ 1/\pi \\ \frac{\pi^2 - 4}{\pi^3} \end{bmatrix}, \quad (3.132)$$

and we recognize the square array as a Hilbert matrix. If we were to consider higher order approximations, the dimension of the Hilbert matrix would increase and its determinant would become exceedingly small, making it very difficult to solve this problem. We should stress that *any* polynomial approximation by least squares leads to this form, not just the particular example we've treated. (Actually, the function being approximated only appears on the right-hand side, and so doesn't influence the array at all. Only the limits of the integration affect the numerical entries in the array.)

Instead of expanding the function as

$$f(x) \approx p(x) = c_0 + c_1x + c_2x^2, \quad (3.133)$$

let's expand it in a more general fashion as

$$f(x) \approx p(x) = \alpha_0 p_0(x) + \alpha_1 p_1(x) + \alpha_2 p_2(x), \quad (3.134)$$

where α_i are coefficients and the $p_i(x)$ are i -th order polynomials. Let's see what conditions we might put on these polynomials to make the normal equations easier to solve. We'll consider the general problem of fitting over the region $[-1, 1]$, and write the error as

$$S = \int_{-1}^1 (f(x) - p_0(x) - p_1(x) - p_2(x))^2 dx, \quad (3.135)$$

partial differentiation with respect to the α_i lead to the normal equations

$$\begin{aligned} \alpha_{-1} \int_0^1 p_0^2(x) dx + \alpha_1 \int_{-1}^1 p_0(x)p_1(x) dx + \alpha_2 \int_{-1}^1 p_0(x)p_2(x) dx \\ = \int_{-1}^1 p_0(x)f(x) dx, \end{aligned} \quad (3.136)$$

$$\begin{aligned} \alpha_0 \int_{-1}^1 p_0(x)p_1(x) dx + \alpha_1 \int_{-1}^1 p_1^2(x) dx + \alpha_2 \int_{-1}^1 p_1(x)p_2(x) dx \\ = \int_{-1}^1 p_1(x)f(x) dx, \end{aligned} \quad (3.137)$$

and

$$\begin{aligned} \alpha_0 \int_{-1}^1 p_0p_2(x) dx + \alpha_1 \int_{-1}^1 p_1(x)p_2(x) dx + \alpha_2 \int_{-1}^1 p_2^2(x) dx \\ = \int_{-1}^1 p_2(x)f(x) dx. \end{aligned} \quad (3.138)$$

Now, these equations for α_i would be easier to solve if we could choose the $p_i(x)$ to make some of these integrals vanish. That is, if

$$\int_{-1}^1 p_i(x)p_j(x) dx = 0, \quad i \neq j, \quad (3.139)$$

then we could immediately solve the normal equations and find that the “best fit” in the least squares sense is obtained when the expansion coefficients are simply

$$\alpha_i = \frac{\int_{-1}^1 p_i(x)f(x) dx}{\int_{-1}^1 p_i^2(x) dx}. \quad (3.140)$$

We should also note that the determination of one coefficient is independent from all the others — once α_1 has been found, it doesn’t change as the expansion is extended to include higher order polynomials. These are remarkable properties, well worth having.

Functions which satisfy Equation (3.139) are called *orthogonal polynomials*, and play several important roles in computational physics. We might define them in a more general fashion, as satisfying

$$\int_a^b w(x)\phi_i(x)\phi_j(x) dx = 0, \quad i \neq j, \quad (3.141)$$

where $[a, b]$ is the region of interest and $w(x)$ is a weighting function. Several well-known functions are actually orthogonal functions, some of which are listed in Table 3.2. Of these, the Legendre, Laguerre, and Hermite polynomials all are of direct physical significance, and so we often find instances in which the physical solution is expanded in terms of these functions. The Chebyshev polynomials have no direct physical significance, but are particularly convenient for approximating functions.

TABLE 3.2 Orthogonal Polynomials

$[a, b]$	$w(x)$	<i>Symbol</i>	<i>Name</i>
$[-1, 1]$	1	$P_n(x)$	Legendre
$[-1, 1]$	$(1 - x^2)^{-1/2}$	$T_n(x)$	Chebyshev I
$[-1, 1]$	$(1 - x^2)^{+1/2}$	$U_n(x)$	Chebyshev II
$[0, \infty)$	e^{-x}	$L_n(x)$	Laguerre
$(-\infty, \infty)$	e^{-x^2}	$H_n(x)$	Hermite

Nonlinear Least Squares

Not all the problems that confront us are linear. Imagine that you have some experimental data that you believe should fit a particular theoretical model. For example, atoms in a gas emit light in a range of wavelengths described by the Lorentzian lineshape function

$$I(\lambda) = I_0 \frac{1}{1 + 4(\lambda - \lambda_0)^2 / \Gamma^2}, \quad (3.142)$$

where λ is the wavelength of the light emitted, λ_0 is the resonant wavelength, Γ is the width of the curve (full width at half maximum), and I_0 is the intensity of the light at $\lambda = \lambda_0$. From measurements of $I(\lambda)$, which necessarily contain experimental noise, we're to determine λ_0 , Γ , and I_0 . (Actually, the intensity is often measured in arbitrary units, and so is unimportant. The position and width of the curve, however, are both significant quantities. The position depends upon the atom emitting the light and hence serves to identify the atom, while the width conveys information concerning atom's environment, e.g., the pressure of the gas.)

Sample data of ambient room light, obtained by an optical multichannel analyzer, are presented in Figure 3.7. A diffraction grating is used to spread the light, which then falls upon a linear array of very sensitive electro-optical elements, so that each element is exposed to some (small) range of wavelengths. Each element then responds according to the intensity of the light falling upon it, yielding an entire spectrum at one time. Note that the data is not falling to zero away from the peak as would be suggested by our theoretical lineshape. This is typical of experimental data — a baseline must

also be established. We thus define our error as

$$S = \sum_{j=1}^N (I_j - B - I(\lambda_j))^2, \quad (3.143)$$

where I_j are the measured intensities. To obtain the normal equations, we minimize S with respect to B , the baseline, and the lineshape parameters λ_0 , Γ , and I_0 , as before. However, $I(\lambda)$ is not a linear function of λ_0 and Γ , so that the normal equations we derive will be *nonlinear* equations, much more difficult than the linear ones we had earlier.

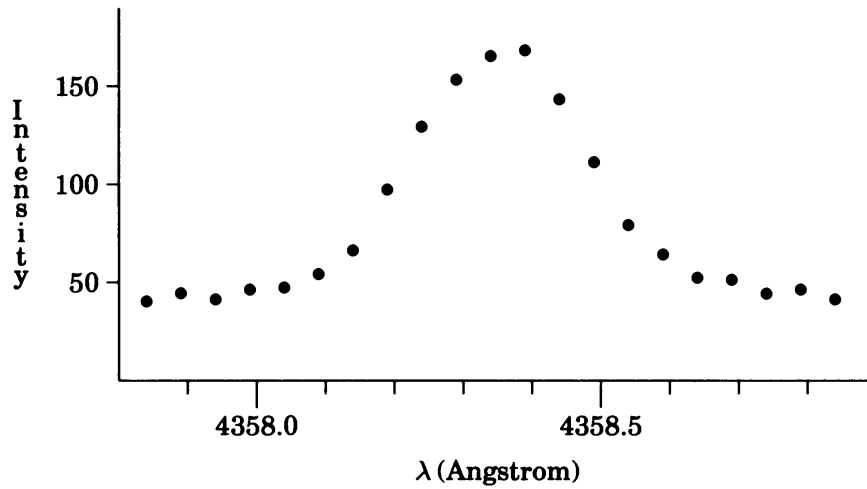


FIGURE 3.7 A spectrum of ambient room light, in arbitrary units of intensity. (Data courtesy L.W. Downes, Miami University.)

Rather than pursue this approach, let's look at the problem from a different perspective. First, let's write the least squares error as

$$S(a_1, a_2, \dots, a_m) = \sum_{k=1}^N (y_k - Y(x_k; a_1, a_2, \dots, a_m))^2, \quad (3.144)$$

where y_k are the data at x_k and $Y(x_k; a_1, a_2, \dots, a_m)$ is the proposed function evaluated at x_k and parameterized by the coefficients a_1, a_2, \dots, a_m . Our goal is to minimize S — why don't we simply vary the parameters a_i until we find a minimum? If we already have initial guesses $a_1^0, a_2^0, \dots, a_m^0$ that lie near a minimum, then S will be approximately quadratic. Let's evaluate S at $a_1^0 + h_1$, a_1^0 , and $a_1^0 - h_1$, holding all the other parameters fixed. The minimum

of a quadratic interpolating polynomial through these points then gives us a better approximation to the minimum of S ,

$$a_1^1 = a_1^0 - \frac{h_1}{2} \frac{S(a_1^0 + h_1, \dots) - S(a_1^0 - h_1, \dots)}{S(a_1^0 + h_1, \dots) - 2S(a_1^0, \dots) + S(a_1^0 - h_1, \dots)}. \quad (3.145)$$

The process is then repeated for a_2, a_3, \dots, a_m . Admittedly, this is a crude procedure, but it works! (Sort of.)

We want to keep the programming as general as possible, so that we don't have to rewrite the routine to solve a different problem. In particular, the routine doesn't need to know much about S — only how many parameters are used, and the value of S at some specific locations. Crucial portions of the code might look something like the following:

```

Subroutine Minimize
double precision a(6),h(6)
integer M
*
* Get initial guesses for the parameters.
*
call init(M,a,h)

call crude(M,a,h)

end
*
*-----
*
Subroutine Init(m,a,h)
double precision a(6),h(6)
integer m,max
parameter ( max = 6)
*
* We need four parameters to determine the lineshape.
*
m = 4
*
* Check if properly dimensioned. If not, will need to
* modify the following array bounds:
*
In routine:      modify arrays:
*
minimize      a, h

```

```

*           crude      a, h, a_plus, a_minus
*           sum        a
*
*           if(m.gt.max) stop 'dimensional error in init'
*
* Initial guesses for the parameters:
*
*           a(1) = 4358.4d0      ! resonant wavelength
*           a(2) = 0.3d0         ! linewidth
*           a(3) = 120.d0        ! intensity
*           a(4) = 40.d0         ! Baseline
*
* Initial values for the step sizes
*
*           h(1) = 0.1d0
*           h(2) = 0.05d0
*           h(3) = 4.d0
*           h(4) = 2.d0
*
*           end
*
* -----
*
*           Subroutine CRUDE (m,a,h)
*           double precision a(6),h(6),a_plus(6),a_minus(6)
*           double precision sp,s0,sm
*           integer i,k,m
*
*           Cycle through each parameter
*
*           DO i = 1, m
*
*           The SUM will be evaluated with the parameters
*           A_PLUS, A, and A_MINUS:
*
*           DO k = 1, m
*             IF(k .eq. i) THEN
*               a_plus(i) = a(i) + h(i)
*               a_minus(i) = a(i) - h(i)
*             ELSE
*               a_plus(k) = a(k)
*               a_minus(k) = a(k)
*             ENDIF
*           END DO

```



```

*
* where
      lambda_0 = a(1)  ! the resonance wavelength
      gamma = a(2)    ! the linewidth
      intensity = a(3)
      baseline = a(4)
*
      SUM = 0.d0
      do i = 1, 21
        lambda = x(i)
*
* evaluate the theoretical curve:
*
      TC = baseline + intensity /
+        (1.d0 + 4.d0*(lambda-lambda_0)**2/gamma**2)
*
* add the square of the difference to the sum:
*
      sum = sum + ( y(i) - TC )**2

      end do
      end

```

As presented, the code will vary each of the parameters once. But, of course, this doesn't assure us of finding a true minimum. In general, we'll need to repeat the process many times, adjusting the parameters each time so that the sum of the errors is decreased. S can be reduced to zero only if the proposed curve fits every data point. Since we assume there is random scatter in the experimental data, this cannot occur. Rather, the error will be reduced to some minimum, determined by the validity of the proposed curve and the "noise" in the data. Modify the code to make repeated calls to CRUDE. As with the root-finder, exercise caution — terminate the process if the error hasn't decreased by half from the previous iteration, and provide a graceful exit after some maximum number — say, ten — iterations.

EXERCISE 3.19

With these modifications, perform a least squares fit to the lineshape data. By the way, can you identify the atom?

Although this crude method works, it can be rather slow to converge. In essence, it's possible for the routine to "walk around" a minimum, getting closer at each step but not moving directly toward it. This happens because

the method does not use any information about the (multidimensional) shape of S to decide in which direction to move — it simply cycles through each parameter in turn. Let's reexamine our approach and see if we can develop a method that is more robust in moving toward the minimum.

Our goal is to determine the point at which

$$\frac{\partial S(a_1, a_2, \dots, a_m)}{\partial a_i} = 0, \quad (3.146)$$

for $i = 1, \dots, m$. (These are just the normal equations we had before.) Let's expand $\partial S / \partial a_i$ about $(a_1^0, a_2^0, \dots, a_m^0)$, keeping only the linear terms,

$$\begin{aligned} \frac{\partial S(a_1, a_2, \dots, a_m)}{\partial a_i} &= \frac{\partial S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i} \\ &+ (a_1 - a_1^0) \frac{\partial}{\partial a_1} \frac{\partial S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i} \\ &+ (a_2 - a_2^0) \frac{\partial}{\partial a_2} \frac{\partial S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i} \\ &+ \dots \\ &+ (a_m - a_m^0) \frac{\partial}{\partial a_m} \frac{\partial S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i}. \end{aligned} \quad (3.147)$$

Defining $\delta_i^0 = a_i - a_i^0$,

$$S_i = \frac{\partial S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i}, \quad (3.148)$$

and

$$S_{ij} = \frac{\partial^2 S(a_1^0, a_2^0, \dots, a_m^0)}{\partial a_i \partial a_j}, \quad (3.149)$$

we arrive at the set of *linear* equations

$$\begin{aligned} S_{11}\delta_1^0 + S_{12}\delta_2^0 + \dots + S_{1m}\delta_m^0 &= -S_1 \\ S_{21}\delta_1^0 + S_{22}\delta_2^0 + \dots + S_{2m}\delta_m^0 &= -S_2 \\ &\vdots \\ S_{m1}\delta_1^0 + S_{m2}\delta_2^0 + \dots + S_{mm}\delta_m^0 &= -S_m. \end{aligned} \quad (3.150)$$

These equations are linear, of course, because we kept only linear terms in the expansion of the partial derivative, Equation (3.147). We can write them

in matrix form as

$$\begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1m} \\ S_{21} & S_{22} & \cdots & S_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ S_{m1} & S_{m2} & \cdots & S_{mm} \end{bmatrix} \begin{bmatrix} \delta_1^0 \\ \delta_2^0 \\ \vdots \\ \delta_m^0 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_m \end{bmatrix}. \quad (3.151)$$

The square array whose elements are the second partial derivatives of S is known as the *Hessian* of S .

Being a linear matrix equation, we can use the subroutine LUsolve to solve for the δ^0 's in Equation (3.151). A better approximation to the location of the minimum will then be

$$a_i^1 = a_i^0 + \delta_i^0, \quad i = 1, \dots, m. \quad (3.152)$$

This is virtually the same process used in finding the roots of equations, except that we now have a set of simultaneous equations to solve. As with root-finding, this process is executed repeatedly, so that for the k -th iteration we would write

$$a_j^k = a_j^{k-1} + \delta_j^{k-1}. \quad (3.153)$$

The process continues until the δ 's are all sufficiently small, or until the iterations fail to significantly reduce S .

This method is nothing more than *Newton's method*, extended to several functions in several independent variables. When close to a minimum, it converges quite rapidly. But if it's started far from the minimum, Newton's method can be slow to converge, and can in fact move to a totally different region of parameter space. In one dimension we placed restrictions on the method so that would not happen, but in multiple dimensions such bounds are not easily placed. (In one dimension, bounds are simply two points on the coordinate, but to bound a region in two dimensions requires a curve, which is much harder to specify.) To increase the likelihood that Newton's method will be successful — unfortunately, we can't guarantee it! — we need a good initial guess to the minimum, which is most easily obtained by using one or two cycles of the CRUDE method before we begin applying Newton's method to the problem. Fundamentally, we're trying to solve a nonlinear problem, a notoriously difficult task full of hidden and disguised subtleties. Nonlinear equations should be approached with a considerable degree of respect and solved with caution.

In Newton's method we need to evaluate the Hessian, the array of second derivatives. This can be done analytically by explicit differentiation of

S . However, it's often more convenient — and less prone to error — to use approximations to these derivatives. From our previous discussion of central difference equations, we can easily find that

$$S_{ii} = \frac{\partial^2 S(\dots, a_i^0, \dots)}{\partial a_i^2} \approx \frac{S(\dots, a_i^0 + h_i, \dots) - 2S(\dots, a_i^0, \dots) + S(\dots, a_i^0 - h_i, \dots)}{h_i^2} \quad (3.154)$$

and

$$S_{ij} = \frac{\partial^2 S(\dots, a_i^0, \dots, a_j^0, \dots)}{\partial a_i^0 \partial a_j^0} \approx \frac{1}{2h_i} \left[\frac{S(\dots, a_i^0 + h_i, \dots, a_j^0 + h_j, \dots) - S(\dots, a_i^0 + h_i, \dots, a_j^0 - h_j, \dots)}{2h_j} - \frac{S(\dots, a_i^0 - h_i, \dots, a_j^0 + h_j, \dots) - S(\dots, a_i^0 - h_i, \dots, a_j^0 - h_j, \dots)}{2h_j} \right]. \quad (3.155)$$

In coding these derivatives, one might consider writing separate expressions for each of them. But this runs counter to our programming philosophy of writing general code, rather than specific examples. Having once written the subroutine, we shouldn't have to change it just because some new problem has 6 parameters instead of 4. With a little thought, we can write a very general routine. The code to calculate the Hessian might look like this:

```

      DOUBLE PRECISION Hessian(6,6),  Ap(6),Am(6),
      +      App(6),Apm(6),Amp(6),Amm(6)
*
*   The arrays have the following contents:
*
*   A      a(1), ...,      a(i), ..., a(m)
*   Ap     a(1), ..., a(i)+h(i),..., a(m)
*   Am     a(1), ..., a(i)-h(i),..., a(m)
*
*   App   a(1),...,a(i)+h(i),..., a(j)+h(j),..., a(m)
*   Apm   a(1),...,a(i)+h(i),..., a(j)-h(j),..., a(m)
*   Amp   a(1),...,a(i)-h(i),..., a(j)+h(j),..., a(m)
*   Amm   a(1),...,a(i)-h(i),..., a(j)-h(j),..., a(m)
*
*       ...
*

```

```

*   Compute the Hessian:
*
      DO i = 1, M
        DO j = i, M
          IF ( i .eq. j ) THEN
            DO k = 1, M
              Ap(k) = A(k)
              Am(k) = A(k)
            END DO
            Ap(i) = A(i) + H(i)
            Am(i) = A(i) - H(i)
            Hessian(i,i)=(sum(Ap) - 2.d0*sum(A)
+              + sum(Am)) / ( h(i) * h(i) )
          ELSE
            DO k = 1, M
              App(k) = A(k)
              Apm(k) = A(k)
              Amp(k) = A(k)
              Amm(k) = A(k)
            END DO
            App(i) = A(i) + H(i)
            App(j) = A(j) + H(j)
            Apm(i) = A(i) + H(j)
            Apm(j) = A(j) - H(j)
            Amp(i) = A(i) - H(i)
            Amp(j) = A(j) + H(j)
            Amm(i) = A(i) - H(i)
            Amm(j) = A(j) - H(j)
          *
            Hessian(i,j) =
+              ( (sum(App)-sum(Apm))/(2.d0*H(j))
+              -(sum(Amp)-sum(Amm))/(2.d0*H(j)) )
+              / ( 2.d0 * H(i) )
            Hessian(j,i) = Hessian(i,j)
          ENDIF
        END DO
      END DO
      ...

```

We've used the array Apm, for example, to store the parameters $a_1, \dots, a_i + h_i, \dots, a_j - h_j, \dots, a_m$. The introduction of these auxiliary arrays ease the evaluation of the second derivatives of the quantity SUM having any number of parameters.

EXERCISE 3.20

Complete the development of the subroutine NEWTON, and apply it to the least squares problem.

References

Interpolation is a standard topic of many numerical analysis texts, and is discussed in several of the works listed at the conclusion of Chapter 2. Gaussian elimination and Crout decomposition are topics from linear algebra, and also are discussed in those texts.

There are many “special functions” of physics, such as gamma functions, Bessel functions, and Legendre, Laguerre, Chebyshev, and Hermite polynomials. These are discussed in many mathematical physics textbooks. I have found the following few to be particularly helpful.

George Arfken, *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

Mary L. Boas, *Mathematical Methods in the Physical Sciences*, John Wiley & Sons, New York, 1983.

Sadri Hassani, *Foundations of Mathematical Physics*, Allyn and Bacon, Boston, 1991.

Chapter 4:

Numerical Integration

Perhaps the most common elementary task of computational physics is the evaluation of integrals. You undoubtedly know many techniques for integrating, and numerical integration should be looked upon as merely another useful method. Certainly, a numerical integration should never be performed if the integral has an analytic solution, except perhaps as a check. You should note, however, that such a check might be used to verify an analytic result, rather than checking the accuracy of the numerical algorithm. It is thus important that we have some brute-force numerical methods to evaluate integrals. Initially, we will consider well-behaved integrals over a finite range, such as

$$I = \int_a^b f(x) dx. \quad (4.1)$$

Eventually we will also treat infinite ranges of integration, and integrands that are not particularly well-behaved. But first, a little Greek history ...

Anaxagoras of Clazomenae

The Age of Pericles, the fifth century B.C., was the zenith of the Athenian era, marked by substantial accomplishments in literature and the arts, and the city of Athens attracted scholars from all of Greece and beyond. From Ionia came Anaxagoras, a proponent of rational inquiry, and a teacher of Pericles. Anaxagoras asserted that the Sun was not a deity, but simply a red-hot glowing stone. Despite the enlightenment of the era, such heresy could not be left unpunished, and he was thrown in prison. (Pericles was eventually able to gain his release.) Plutarch tells us that while Anaxagoras was imprisoned, he occupied himself by attempting to square the circle. This is the first record of this problem, one of the three “classic problems of antiquity”: to square the circle, to double a cube, and to trisect an angle. As now understood, the problem is to construct a square, exactly equal in area to the circle, using only compass and straightedge. This problem was not finally “solved” until 1882,

when it was proved to be impossible!

The fundamental goal, of course, is to determine the area of the circle. For over two thousand years this problem occupied the minds of the great scholars of the world. In partial payment of those efforts, we refer to the numerical methods of integration, and hence finding the area bounded by curves, as integration by *quadrature*. That is, integration can be thought of as finding the edglength of the square, i.e., four-sided regular polygon *ergo quad*, with an area equal to that bounded by the original curve.

Primitive Integration Formulas

It is instructive, and not very difficult, to derive many of the commonly used integration formulas. The basic idea is to evaluate the function at specific locations, and to use those function evaluations to approximate the integral. We thus seek a *quadrature* formula of the form

$$I \approx \sum_{i=0}^N W_i f_i, \quad (4.2)$$

where x_i are the evaluation points, $f_i = f(x_i)$, W_i is the weight given the i -th point, and $N + 1$ is the number of points evaluated. To keep things simple, we will use equally spaced evaluation points separated by a distance h . Let's begin by deriving a *closed* formula, one that uses function evaluations at the endpoints of the integration region. The simplest formula is thus

$$I \approx W_0 f_0 + W_1 f_1, \quad (4.3)$$

where $x_0 = a$ and $x_1 = b$, the limits of the integration. We want this approximation to be useful for a variety of integrals, and so we'll require that it be *exact* for the simplest integrands, $f(x) = 1$ and $f(x) = x$. Since these are the first two terms in a Taylor series expansion of any function, this approximation will converge to the exact result as the integration region is made smaller, for any $f(x)$ that has a Taylor series expansion. We thus require that

$$\int_{x_0}^{x_1} 1 \, dx = x_1 - x_0 = W_0 + W_1,$$

and

$$\int_{x_0}^{x_1} x \, dx = \frac{x_1^2 - x_0^2}{2} = W_0 x_0 + W_1 x_1. \quad (4.4)$$

This is simply a set of two simultaneous equations in two unknowns, W_0 and W_1 , easily solved by the methods of linear algebra to give

$$W_0 = W_1 = \frac{x_1 - x_0}{2}. \quad (4.5)$$

Writing $h = b - a$, the approximation is then

$$\int_{x_0}^{x_1} f(x) dx \approx \frac{h}{2}(f_0 + f_1), \quad (4.6)$$

the so-called trapezoid rule. For the sake of completeness we'll exhibit these formulas as equalities, using Lagrange's expression for the remainder term in the Taylor series expansion, Equation (2.7), so that we have

$$\int_{x_0}^{x_1} f(x) dx = \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12}f^{[2]}(\xi), \quad (4.7)$$

where ξ is some point within the region of integration. Of course, we're not limited to $N = 1$; for three points, we find the equations

$$\int_{x_0}^{x_2} 1 dx = x_2 - x_0 = W_0 + W_1 + W_2,$$

$$\int_{x_0}^{x_2} x dx = \frac{x_2^2 - x_0^2}{2} = W_0x_0 + W_1x_1 + W_2x_2,$$

and

$$\int_{x_0}^{x_2} x^2 dx = \frac{x_2^3 - x_0^3}{3} = W_0x_0^2 + W_1x_1^2 + W_2x_2^2. \quad (4.8)$$

Solving for the weights, we are led to Simpson's rule,

$$\int_{x_0}^{x_2} f(x) dx = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f^{[4]}(\xi). \quad (4.9)$$

Continuing in this fashion, with 4 points we are led to Simpson's three-eighths rule,

$$\int_{x_0}^{x_3} f(x) dx = \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80}f^{[4]}(\xi), \quad (4.10)$$

while with five points we find Boole's rule

$$\int_{x_0}^{x_4} f(x) dx = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945}f^{[6]}(\xi). \quad (4.11)$$

These integration formulas all require that $f(x)$ be expressed as a polynomial, and could have been derived by fitting $f(x)$ to an approximating polynomial and integrating that function exactly. As more points are incorporated in the integration formula, the quadrature is exact for higher degree polynomials. This process could be continued indefinitely, although the quadratures become increasingly complicated.

Composite Formulas

As an alternative to higher order approximations, we can divide the total integration region into many segments, and use a low order, relatively simple quadrature over each segment. Probably the most common approach is to use the trapezoid rule, one of the simplest quadratures available. Dividing the region from x_0 to x_N into N segments of width h , we can apply the trapezoid rule to each segment to obtain the composite trapezoid rule

$$\begin{aligned}\int_{x_0}^{x_N} f(x) dx &\approx \frac{h}{2}(f_0 + f_1) + \frac{h}{2}(f_1 + f_2) + \cdots + \frac{h}{2}(f_{N-1} + f_N) \\ &= h\left(\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{N-1} + \frac{f_N}{2}\right).\end{aligned}\quad (4.12)$$

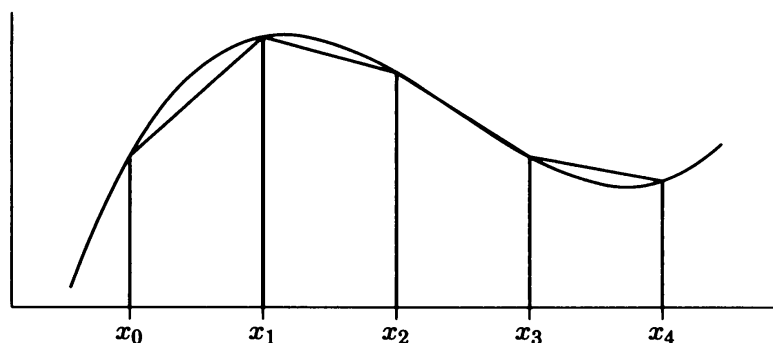


FIGURE 4.1 Constructing a composite formula from the trapezoid rule.

This is equivalent to approximating the actual integrand by a series of straight line segments, as in Figure 4.1, and integrating; such a fit is said to be piecewise linear. A similar procedure yields a composite Simpson's rule,

$$\int_{x_0}^{x_N} f(x) dx \approx \frac{h}{3}(f_0 + 4f_1 + 2f_2 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N), \quad (4.13)$$

where h is again the distance between successive function evaluations. This is equivalent to a piecewise quadratic approximation to the function, and hence

is more accurate than the composite trapezoid rule using the same number of function evaluations. Piecewise cubic and quartic approximations are obviously available by making composite versions of the primitive integration formulas in Equations (4.10) and (4.11).

■ EXERCISE 4.1

Evaluate the integral $\int_0^\pi \sin x \, dx$ using approximations to the integrand that are piecewise linear, quadratic, and quartic. With N intervals, and hence $N + 1$ points, evaluate the integral for $N = 4, 8, 16, \dots, 1024$, and compare the accuracy of the methods.

Errors ... and Corrections

Let's rederive the trapezoid rule. Earlier, we noted that we can derive quadrature formulas by expanding the integrand in a Taylor series and integrating. Let's try that: consider the integral of Equation (4.1), and expand the function $f(x)$ in a Taylor series about $x = a$:

$$\begin{aligned} \int_a^b f(x) \, dx &= \int_a^b \left[f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!} f''(a) \right. \\ &\quad \left. + \frac{(x-a)^3}{3!} f'''(a) + \frac{(x-a)^4}{4!} f^{[4]}(a) + \dots \right] dx \\ &= hf(a) + \frac{h^2}{2!} f'(a) + \frac{h^3}{3!} f''(a) + \frac{h^4}{4!} f'''(a) + \frac{h^5}{5!} f^{[4]}(a) + \dots \end{aligned} \quad (4.14)$$

But we could just as easily have expanded about $x = b$, and would have found

$$\begin{aligned} \int_a^b f(x) \, dx &= \int_a^b \left[f(b) + (x-b)f'(b) + \frac{(x-b)^2}{2!} f''(b) \right. \\ &\quad \left. + \frac{(x-b)^3}{3!} f'''(b) + \frac{(x-b)^4}{4!} f^{[4]}(b) + \dots \right] dx \\ &= hf(b) - \frac{h^2}{2!} f'(b) + \frac{h^3}{3!} f''(b) - \frac{h^4}{4!} f'''(b) + \frac{h^5}{5!} f^{[4]}(b) + \dots \end{aligned} \quad (4.15)$$

The symmetry between these two expressions is quite evident. A new expres-

sion for the integral can be obtained by simply adding these two equations,

$$\begin{aligned} \int_a^b f(x) dx = & \frac{h}{2} [f(a) + f(b)] + \frac{h^2}{4} [f'(a) - f'(b)] + \frac{h^3}{12} [f''(a) + f''(b)] \\ & + \frac{h^4}{48} [f'''(a) - f'''(b)] + \frac{h^5}{240} [f^{[4]}(a) + f^{[4]}(b)] + \dots \end{aligned} \quad (4.16)$$

As it stands, this expression is no better or worse than the previous two, but it's leading us to an interesting and useful result. We see that the odd- and even-order derivatives enter this expression somewhat differently, as either a difference or sum of derivatives at the endpoints. Let's see if we can eliminate the even-order ones — the motivation will be clear in just a bit. Making a Taylor series expansion of $f'(x)$ about the point $x = a$, we have

$$f'(x) = f'(a) + (x - a)f''(a) + \frac{(x - a)^2}{2}f'''(a) + \frac{(x - a)^3}{6}f^{[4]}(a) + \dots \quad (4.17)$$

In particular, at $x = b$,

$$f'(b) = f'(a) + hf''(a) + \frac{h^2}{2}f'''(a) + \frac{h^3}{6}f^{[4]}(a) + \dots \quad (4.18)$$

And of course, we could have expanded about $x = b$, and have found

$$f'(a) = f'(b) - hf''(b) + \frac{h^2}{2}f'''(b) - \frac{h^3}{6}f^{[4]}(b) + \dots \quad (4.19)$$

These expressions can be combined to yield

$$\begin{aligned} f''(a) + f''(b) = & \frac{2}{h} [f'(b) - f'(a)] - \frac{h}{2} [f'''(a) - f'''(b)] \\ & - \frac{h^2}{6} [f^{[4]}(a) + f^{[4]}(b)] + \dots \end{aligned} \quad (4.20)$$

We could also expand the $f'''(x)$ about $x = a$ and $x = b$ to find

$$f^{[4]}(a) + f^{[4]}(b) = \frac{2}{h} [f'''(b) - f'''(a)] + \dots \quad (4.21)$$

Using Equations (4.20) and (4.21), the even-order derivatives can now be eliminated from Equation (4.16), so that we have

$$\int_a^b f(x) dx = \frac{h}{2} [f(a) + f(b)] + \frac{h^2}{12} [f'(a) - f'(b)] - \frac{h^4}{720} [f'''(a) - f'''(b)] + \dots \quad (4.22)$$

The importance of Equation (4.22) is not that the even derivatives are missing, but that the derivatives that are present appear in differences. A new composite formula can now be developed by dividing the integration region from a to b into many smaller segments, and applying Equation (4.22) to each of these segments. Except at the points a and b , the odd-order derivatives contribute to two adjacent segments, once as the left endpoint and once as the right. But they contribute *with different signs*, so that their contributions to the total integral cancel! We thus find the *Euler–McClaurin* integration rule,

$$\begin{aligned} \int_{x_0}^{x_N} f(x) dx = & h \left(\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{N-1} + \frac{f_N}{2} \right) \\ & + \frac{h^2}{12} [f'_0 - f'_N] - \frac{h^4}{720} [f'''_0 - f'''_N] + \cdots \end{aligned} \quad (4.23)$$

This is simply the trapezoid rule, Equation (4.12), with correction terms, and so gives us the error incurred when using the standard composite trapezoid rule. When the integrand is easily differentiated, the Euler–McClaurin integration formula is far superior to other numerical integration schemes. A particularly attractive situation arises when the derivatives vanish at the limits of the integration, so that the “simple” trapezoid rule can yield surprisingly accurate results.

EXERCISE 4.2

Reevaluate the integral $\int_0^\pi \sin x \, dx$ using the trapezoid rule with first one and then two correction terms, and compare to the previous calculations for $N = 4, 8, \dots, 1024$ points. Since the integrand is relatively simple, analytic expressions for the correction terms should be used.

Romberg Integration

Although the Euler–McClaurin formula is very powerful when it can be used, the more common situation is that the derivatives are not so pleasant to evaluate. However, from Euler–McClaurin we know how the error behaves, and so we can apply Richardson extrapolation to obtain an improved approximation to the integral.

Let's denote the result obtained by the trapezoid rule with $n = 2^m$ intervals as $T_{m,0}$, which is known to have an error proportional to h^2 . Then, $T_{m+1,0}$, obtained with an interval half as large, will have one-fourth the error

of the previous approximation. We can thus combine these two approximations to eliminate this leading error term, and obtain

$$T_{m+1,1} = \frac{4T_{m+1,0} - T_{m,0}}{3}. \quad (4.24)$$

(Though not at all obvious, you might want to convince yourself that this is simply Simpson's rule.) Of course, we only eliminated the *leading* term — $T_{m+1,1}$ is still in error, but proportional to h^4 . If we halve the interval size again, the next approximation will have one-sixteenth the error of this approximation, which can then be eliminated to yield

$$T_{m+2,2} = \frac{16T_{m+2,1} - T_{m+1,1}}{15}. \quad (4.25)$$

(Again not obvious, this is Boole's rule.) In this way, a triangular array of increasingly accurate results can be obtained, with the general entry in the array given by

$$T_{m+k,k} = \frac{4^k T_{m+k,k-1} - T_{m+k-1,k-1}}{4^k - 1}. \quad (4.26)$$

For moderately smooth functions, this *Romberg integration scheme* yields very good results. As we found with Richardson extrapolation of derivatives, there are decreasing benefits associated with higher and higher extrapolations, so that k should probably be kept less than 4 or 5.

The code fragment developed earlier for Richardson extrapolation is directly applicable to the present problem, using the trapezoid rule approximation to the integral as the function being extrapolated. The composite trapezoid rule can itself be simplified, since all (interior) function evaluations enter with the same weight and all the points used in calculating $T_{m,0}$ were already used in calculating $T_{m-1,0}$. In fact, it's easy to show that

$$\int_a^b f(x) dx \approx T_{m,0} = \frac{1}{2}T_{m-1,0} + h \sum_{i=1,3,\dots}^{2^m-1} f(a + ih),$$

$$h = \frac{b-a}{2^m}, \quad (4.27)$$

where the sum over i is over the points *not* included in $T_{m-1,0}$.

EXERCISE 4.3

Write a computer code to perform Romberg integration, obtaining 8-figure accuracy in the calculation of the integral, as determined by a

relative error check. Your code should include a “graceful exit” if convergence hasn’t been obtained after a specified number of halvings, say, if $m \geq 15$. The dimensions used in the previous code fragment should be adjusted appropriately. Test your code on the integral

$$\int_0^{\pi/2} \frac{d\theta}{1 + \cos \theta}.$$

Diffraction at a Knife’s Edge

In optics, we learn that light “bends around” objects, i.e., it exhibits diffraction. Perhaps the simplest case to study is the bending of light around a straightedge. In this case, we find that the intensity of the light varies as we move away from the edge according to

$$I = 0.5I_0 \left\{ [C(v) + 0.5]^2 + [S(v) + 0.5]^2 \right\}, \quad (4.28)$$

where I_0 is the intensity of the incident light, v is proportional to the distance moved, and $C(v)$ and $S(v)$ are the Fresnel integrals

$$C(v) = \int_0^v \cos(\pi w^2/2) dw \quad (4.29)$$

and

$$S(v) = \int_0^v \sin(\pi w^2/2) dw. \quad (4.30)$$

EXERCISE 4.4

Numerically integrate the Fresnel integrals, and thus evaluate I/I_0 as a function of v . Plot your results.

A Change of Variables

Well, now we have a super-duper method of integrating ... or do we? Consider the integral

$$I = \int_{-1}^1 \sqrt{1 - x^2} dx, \quad (4.31)$$

displayed in Figure 4.2.

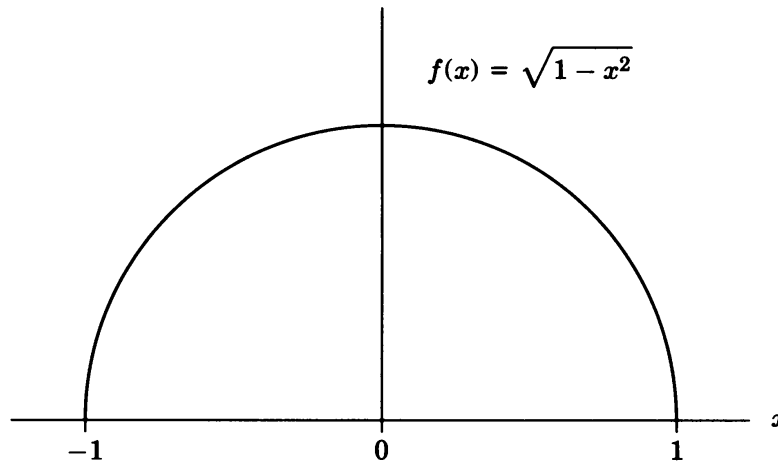


FIGURE 4.2 A simple-looking integral.

The integral doesn't look like it should give us much trouble, yet when we try to do Romberg integration, we generate the following table:

m	$T_{m,0}$	$T_{m,1}$	$T_{m,2}$	$T_{m,3}$
0	0.00000 000			
1	1.00000 000	1.33333 333		
2	1.36602 540	1.48803 387	1.49834 724	
3	1.49785 453	1.54179 758	1.54538 182	1.54612 841
4	1.54490 957	1.56059 458	1.56184 772	1.56210 908
5	1.56162 652	1.56719 883	1.56763 912	1.56773 104
6	1.56755 121	1.56952 611	1.56968 126	1.56971 368
7	1.56964 846	1.57034 754	1.57040 230	1.57041 375
8	1.57039 040	1.57063 771	1.57065 705	1.57066 110
9	1.57065 279	1.57074 026	1.57074 709	1.57074 852
10	1.57074 558	1.57077 650	1.57077 892	1.57077 943

It's clear that *none* of the approximations are converging very rapidly — What went wrong?

Well, let's see ... The Euler–McClaurin formula gives us an idea of what the error should be, in terms of the derivatives of the integrand at the endpoints. And for this integrand, these derivatives are *infinite*! No wonder Romberg integration failed!

Before trying to find a “cure,” let's see if we can find a diagnostic that will indicate when the integration is failing. Romberg integration works, when it works, because the error in the trapezoid rule is quartered when the step size is halved — that seems simple enough to check. Using the difference between $T_{m+1,0}$ and $T_{m,0}$ as an indication of the error “at this step,” we can

evaluate the ratio of errors at consecutive steps. In order for the method to work, this ratio should be about 4:

$$R_m = \frac{T_{m-1,0} - T_{m,0}}{T_{m,0} - T_{m+1,0}} \approx 4. \quad (4.32)$$

Using the trapezoid approximations given in the table, we compute the ratios to be

m	R_m
1	2.73205
2	2.77651
3	2.80159
4	2.81481
5	2.82157

These ratios are obviously not close to 4, and the integration fails. The reason, of course, is that the Taylor series expansion, upon which all this is based, has broken down — at the endpoints, the derivative of the function is infinite! Therefore, the approximation for the integral in the first and last intervals is terrible! The total integral converges, slowly, only because smaller and smaller steps are being taken, so that the contributions from the first and last intervals are correspondingly reduced. In essence, you are required to take a step size sufficiently small that the error incurred in the end intervals has been reduced to a tolerable level. *All those function evaluations in the interior of the integration region have done nothing for you!* What a waste.

This problem is crying out to you, *use a different spacing!* Put a lot of evaluations at the ends, if you need them there, and not so many in the middle. Mathematically, this is accomplished by a change of variables. This is, of course, a standard technique of analytic integration and a valuable tool in numerical work as well. The form of the integrand suggests that we try a substitution of the form

$$x = \cos \theta, \quad (4.33)$$

so that the integral becomes

$$I = \int_{-1}^1 \sqrt{1-x^2} dx = \int_0^\pi \sin^2 \theta d\theta. \quad (4.34)$$

Using the Romberg integrator to evaluate this integral, we generate the following results:

m	$T_{m,0}$	$T_{m,1}$	$T_{m,2}$	$T_{m,3}$
0	0.00000 000			
1	1.57079 633	2.09439 510		
2	1.57079 633	1.57079 633	1.53588 974	
3	1.57079 633	1.57079 633	1.57079 633	1.57135 040
4	1.57079 633	1.57079 633	1.57079 633	1.57079 633

At first blush, these results are more surprising than the first ones — *only one nonzero function evaluation was required to yield the exact result!* Well, one thing at a time. First, it's only coincidence that the result is exact — in general, you have to do *some* work to get a good result. However, we shouldn't have to work too hard — *look at the derivatives at the endpoints!* For this integrand, all those derivatives are zero, so that Euler–McClaurin tells us that the trapezoid rule is good enough, once h is sufficiently small.

As a practical matter, we need to do two things to the Romberg computer code. First, the ratios R_m should be evaluated, to verify that the method is working correctly. The ratio will not be *exactly* 4, but it should be close, say, within 10%. If the ratio doesn't meet this requirement, the program should exit and print the relevant information. (Of course, if the requirement *is* met, there's no reason to print any of this information. However, a message indicating that the test was performed and the requirement met should be printed.) The second thing relates to the last table we generated. All the results were exact, except the first ones in each column. In general, we should provide the integrator with a reasonable approximation before we begin the extrapolation procedure. Delaying the extrapolation a few iterations, say, until $m = 2$ or 3, simply avoids the spurious approximations generated early in the process and has little effect on the ultimate accuracy of the integration.

EXERCISE 4.5

Modify your Romberg integrator, and use it to evaluate the elliptic integral

$$I = \int_{-1}^1 \sqrt{(1-x^2)(2-x)} \, dx.$$

First, test your diagnostic by trying to evaluate the integral in its present form. If the diagnostic indicates that Romberg integration is failing, perform a change of variables and integrate again.

The “Simple” Pendulum

A standard problem of elementary physics is the motion of the simple pendu-

lum, as shown in Figure 4.3. From Newton’s laws, or from either Lagrange’s or Hamilton’s formulations, the motion is found to be described by the differential equation

$$ml \frac{d^2\theta}{dt^2} = -mg \sin \theta. \quad (4.35)$$

The mass doesn’t enter the problem, and the equation can be written as

$$\ddot{\theta} = -\frac{g}{l} \sin \theta \quad (4.36)$$

where we’ve written the second time derivative as $\ddot{\theta}$. For *small amplitudes*, $\sin \theta \approx \theta$, so that the motion is described approximately by the equation

$$\ddot{\theta} = -\frac{g}{l} \theta, \quad (4.37)$$

which has as its solution

$$\theta(t) = \theta_0 \cos \sqrt{\frac{g}{l}} t, \quad (4.38)$$

for the case of the motion beginning at $t = 0$ with the pendulum motionless at $\theta = \theta_0$.

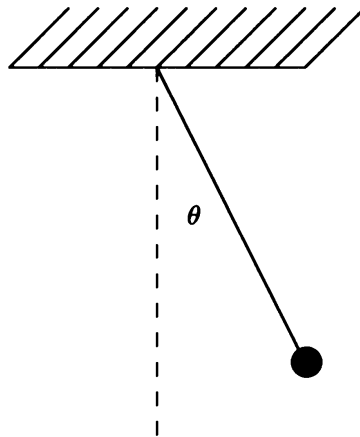


FIGURE 4.3 The simple pendulum.

But what about the *real* motion? For large amplitude (how large?), we should expect this *linearized* description to fail. In particular, this description tells us that the period of the oscillation is independent of the amplitude, or *isochronous*, which seems unlikely. Let’s see if we can do better.

Now, we could attempt an improved description by applying perturbation theory — that is, by starting with the linearized description and finding

corrections to it. In the days before computers, this was not only the approach of choice, it was virtually the only approach available! Today, of course, we have computers, and ready access to them. We *could* simply solve the differential equation describing the exact motion, directly. And in Chapter 5, that's exactly what we'll do! But to *always* jump to the numerical solution of the differential equation is to be just as narrow-minded and shortsighted as insisting on applying perturbation theory to every problem that comes along. There are many ways to approach a problem, and we often learn *different things* as we explore different approaches. This lesson, learned when analytic methods were all we had, is just as valuable now that we have new tools to use. The computational approach to physics is most successful when numerical methods are used to complement analytic ones, not simply replace them.

Starting with Equation (4.36), multiply both sides by $\dot{\theta}$ and integrate to obtain

$$\frac{1}{2}\dot{\theta}^2 = \frac{g}{l} \cos \theta + C, \quad (4.39)$$

where C is a constant of integration, to be determined from the initial conditions: for $\dot{\theta} = 0$ at $t = 0$, we have $C = -g/l \cos \theta_0$. Solving for $\dot{\theta}$, we find

$$\dot{\theta} = \frac{d\theta}{dt} = \sqrt{\frac{2g}{l}} \sqrt{\cos \theta - \cos \theta_0}, \quad (4.40)$$

or

$$dt = \sqrt{\frac{l}{2g}} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}. \quad (4.41)$$

Now the total period, T , is just four times the time it takes the pendulum to travel from $\theta = 0$ to $\theta = \theta_0$, so that

$$T = 4\sqrt{\frac{l}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}. \quad (4.42)$$

This is an elliptic integral of the first kind, and is an exact result for the period of the pendulum. But of course, it remains to *evaluate* the integral. In particular, the integrand is a little unpleasant at the upper limit of integration.

Let's convert this integral into the "standard" form, originated by Legendre, using the identity

$$\cos \theta - \cos \theta_0 = 2\left(\sin^2 \frac{\theta_0}{2} - \sin^2 \frac{\theta}{2}\right) \quad (4.43)$$

and the substitution

$$\sin \xi = \frac{\sin \frac{\theta}{2}}{\sin \frac{\theta_0}{2}}. \quad (4.44)$$

We then find that

$$T = 4\sqrt{\frac{l}{g}} K(\sin \frac{\theta_0}{2}), \quad (4.45)$$

where

$$K(k) = \int_0^{\pi/2} \frac{d\xi}{\sqrt{1 - k^2 \sin^2 \xi}} \quad (4.46)$$

is the *complete elliptic integral of the first kind*. Clearly, this integrand is much nicer than the previous one. The most general elliptic integral of the first kind is parametrically dependent upon the upper limit of the integration,

$$F(k, \phi) = \int_0^{\phi} \frac{d\xi}{\sqrt{1 - k^2 \sin^2 \xi}}. \quad (4.47)$$

From these definitions, we clearly have that $K(k) = F(k, \frac{\pi}{2})$.

The real value of a “standard form” is that you might find values for it tabulated somewhere. You could then check your integration routine by computing the standard integral for the values tabulated, before you solved the particular problem of your interest. For example, the following table is similar to one found in several reference books of mathematical functions.

$$K(k) = \int_0^{\pi/2} \frac{d\xi}{\sqrt{1 - k^2 \sin^2 \xi}}$$

$\sin^{-1} k$	$K(k)$
0°	1.57079 63270
10°	1.58284 28043
20°	1.62002 58991
30°	1.68575 03548
40°	1.78676 91349
50°	1.93558 10960
60°	2.15651 56475
70°	2.50455 00790
80°	3.15338 52519
90°	∞

You might want to verify these values before returning your attention to the motion of the simple pendulum.

EXERCISE 4.6

Calculate the period of the simple pendulum, using whatever method you feel appropriate to evaluate the integral. Be prepared to justify your choice. Produce a table of results, displaying the calculated period versus the amplitude. For what values of the amplitude does the period differ from $2\pi\sqrt{l/g}$ by more than 1%? Is there a problem at $\theta_0 = 180^\circ$? Why?

Legendre was thoroughly absorbed by these integrals, and developed a second and third standard form. Of some interest to physics is the elliptic integral of the second kind,

$$E(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \xi} \, d\xi, \quad (4.48)$$

which arises in determining the length of an elliptic arc.

Improper Integrals

Integrals of the form

$$I = \int_0^\infty f(x) \, dx \quad (4.49)$$

are certainly not uncommon in physics. It is possible that this integral diverges; that is, that $I = \infty$. If this is the case, don't ask your numerical integrator for the answer! But often these integrals do exist, and we need to be able to handle them. Many techniques are available to us; however, none of them *actually* tries to integrate numerically all the way to infinity. That is, if the integral is

$$I = \int_0^\infty x^2 e^{-x} \, dx, \quad (4.50)$$

it might be tempting to integrate up to some large number A , and then go a little farther to A' , and stop if the integrals aren't much different. But "how much farther" is enough? — infinity is a long way off! It's better to use a different approach, one which in some way accounts for the infinite extent of the integration domain.

One such approach is to split the integration region into two segments,

$$\int_0^\infty f(x) \, dx = \int_0^a f(x) \, dx + \int_a^\infty f(x) \, dx. \quad (4.51)$$

The first integral is over a finite region and can be performed by a standard numerical method. The appropriate value for a will depend upon the particular integrand, and how the second integral is to be handled. For example, it can be mapped into a finite region by a change of variables such as

$$x \rightarrow 1/y,$$

so that the integral becomes

$$\int_a^\infty f(x) dx = \int_0^{1/a} \frac{f(y^{-1})}{y^2} dy. \quad (4.52)$$

If the change of variables hasn't introduced an additional singularity into the integrand, this integral can often be evaluated by one of the standard numerical methods.

EXERCISE 4.7

Evaluate the integral

$$I = \int_0^\infty \frac{dx}{1+x^2}$$

to 8 significant digits. (Let $a = 1$, and break the interval into two domains.)

Note that $x \rightarrow 1/y$ is not the only substitution that might be used — for a particular $f(x)$ it might be better to choose $x \rightarrow e^{-y}$, for example. The substitution

$$x \rightarrow \frac{1+y}{1-y} \quad (4.53)$$

is interesting in this regard as it transforms the interval $[0, \infty]$ into $[-1, 1]$, which is particularly convenient for Gauss–Legendre integration, to be discussed latter. The goal is to use a transform that maps the semi-infinite region into a finite one while yielding an integrand that we can evaluate.

EXERCISE 4.8

Using the transformation

$$x \rightarrow \frac{y}{1-y}, \quad (4.54)$$

transform the integral

$$I = \int_0^\infty \frac{x dx}{(1+x)^4} \quad (4.55)$$

into one over a finite domain, and evaluate.

There's another approach we can take to the evaluation of an integral: expand all or part of it in an infinite series. Let's imagine that we need to evaluate the integral

$$I = \int_0^{\infty} \frac{dx}{(1+x)\sqrt{x}}, \quad (4.56)$$

and have broken it into two domains. We first consider the integral

$$I_2 = \int_a^{\infty} \frac{dx}{(1+x)\sqrt{x}}, \quad (4.57)$$

where we have chosen a to be much larger than 1. Then we can use the binomial theorem to write

$$\frac{1}{1+x} = \frac{1}{x(1+\frac{1}{x})} = \frac{1}{x} \left(1 - \frac{1}{x} + \frac{1}{x^2} - \dots\right). \quad (4.58)$$

Substituting this into the integral, we find

$$\begin{aligned} \int_a^{\infty} \frac{dx}{(1+x)\sqrt{x}} &= \int_a^{\infty} (x^{-\frac{3}{2}} - x^{-\frac{5}{2}} + x^{-\frac{7}{2}} - \dots) dx \\ &= 2a^{-\frac{1}{2}} - \frac{2}{3}a^{-\frac{3}{2}} + \frac{2}{5}a^{-\frac{5}{2}} - \dots. \end{aligned} \quad (4.59)$$

The series will converge, by virtue of the convergence of the binomial expansion, for any $a > 1$, although it clearly converges faster for larger a . Series expansions can be extremely useful in situations such as these, but we must always be certain that we are using the series properly. In particular, the series will only converge if it's being used within its domain of convergence.

So, all we need to do now is to perform the integral from zero to a , but there seems to be a small problem ...

$$I_1 = \int_0^a \frac{dx}{(1+x)\sqrt{x}} \quad (4.60)$$

— the integrand is infinite at one endpoint. Before proceeding, we should probably convince ourselves that the integral is finite. This can be done by noting that

$$\frac{1}{1+x} \leq 1, \quad x \geq 0,$$

so that

$$\int_0^a \frac{dx}{(1+x)\sqrt{x}} \leq \int_0^a \frac{dx}{\sqrt{x}} = 2\sqrt{a}. \quad (4.61)$$

We thus have a finite upper bound on the integral, so the integral itself must be finite. To evaluate the integral, we might try a power series expansion such as

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + \cdots. \quad (4.62)$$

However, we must note that this series converges *if and only if* $x < 1$. Since we've already specified that $a > 1$, this series cannot be used over the entire integration region. However, we could break the interval again, into one piece near the origin and the other piece containing everything else:

$$\int_0^a \frac{dx}{(1+x)\sqrt{x}} = \int_0^\epsilon \frac{dx}{(1+x)\sqrt{x}} + \int_\epsilon^a \frac{dx}{(1+x)\sqrt{x}}. \quad (4.63)$$

This is a very reasonable approach to take — the idea of isolating the difficult part into one term upon which to concentrate is a very good tactic, one that we often use. In some cases, it might happen that you not even need to do the difficult part. For example, in this case you might evaluate the second integral for increasingly smaller ϵ and extrapolate to the limit of $\epsilon \rightarrow 0$.

Let's try something totally different from the power series method — just *subtract* the singularity away! We've already discovered that near zero the integrand behaves as $1/\sqrt{x}$, which we were able to integrate to yield an upper bound. So, let's simply write

$$\begin{aligned} \int_0^a \frac{dx}{(1+x)\sqrt{x}} &= \int_0^a \left[\frac{1}{\sqrt{x}} + \frac{1}{(1+x)\sqrt{x}} - \frac{1}{\sqrt{x}} \right] dx \\ &= \int_0^a \frac{dx}{\sqrt{x}} + \int_0^a \frac{-x dx}{(1+x)\sqrt{x}} \\ &= 2\sqrt{a} - \int_0^a \frac{\sqrt{x} dx}{1+x}. \end{aligned} \quad (4.64)$$

The desired integral is thus expressed as an integrated term containing the difficult part, plus a difference expression that can be integrated numerically.

EXERCISE 4.9

Evaluate the integral

$$I = \int_0^2 \frac{dx}{(1+x)\sqrt{x}}$$

by subtracting the singularity away.

If we can add and subtract terms in an integrand, we can also multiply and divide them. In general, we might write

$$\int f(x) dx = \int \frac{f(x)}{g(x)} g(x) dx. \quad (4.65)$$

This hasn't gained us much, unless $g(x)$ is chosen so that $g(x) dx = dy$. If this is so, then we can make the substitution $x \rightarrow y$ to arrive at the integral

$$\int \frac{f(y)}{g(y)} dy. \quad (4.66)$$

Since we are free to choose g , we will choose it so that the integrand in this expression is “nicer” than just $f(x)$. This, of course, is nothing more than a rigorous mathematical description of a change of variables, with the function g being the Jacobian of the transformation. In our example, we might choose $g(x) = 1/\sqrt{x}$, so that

$$\int_0^a \frac{dx}{(1+x)\sqrt{x}} = \int_0^a \left(\frac{\sqrt{x}}{(1+x)\sqrt{x}} \right) \left(\frac{dx}{\sqrt{x}} \right). \quad (4.67)$$

We now have $dy = dx/\sqrt{x}$, or $y = 2\sqrt{x}$. Note that we are choosing g to make the integrand nice, and from that *deriving* what the new variable should be—virtually the reverse of the usual change of variable procedure. Making the substitution, we find

$$\int_0^a \frac{dx}{(1+x)\sqrt{x}} = \int_0^{2\sqrt{a}} \frac{1}{1+y^2/4} dy. \quad (4.68)$$

Once again, we find that the transformed integrand will be easy to integrate, which of course was the motivation for transforming in the first place.

EXERCISE 4.10

Evaluate the integral

$$I = \int_0^2 \frac{dx}{(1+x)\sqrt{x}}$$

by changing the variable of integration.

From time to time, you'll have to evaluate a function that superficially appears divergent. An example of such a situation is the integral

$$\int_0^\pi \frac{\sin x}{x} dx.$$

The integral exists, and in fact the function is finite at every point. However, if you simply ask the computer to evaluate the integrand at $x = 0$, you'll have problems with "zero divided by zero." Clearly, either a series expansion of $\sin x$ or the use of L'Hospital's rule will remove the difficulty — but that's your job, not the computer's!

EXERCISE 4.11

Using the integration tools we've discussed, evaluate the integral

$$\int_0^1 \frac{\cos x}{\sqrt{x}} dx.$$

The Mathematical Magic of Gauss

Earlier we performed numerical integration by repeated use of the trapezoid rule, using the composite formula

$$\int_{x_0}^{x_N} f(x) dx = h \left[\frac{f_0}{2} + f_1 + \cdots + f_{N-1} + \frac{f_N}{2} \right], \quad (4.69)$$

where $h = (x_N - x_0)/N$ and the x_m , where the function is being evaluated, are evenly spaced on the interval $[a, b]$. That is, the total integration region has been divided into N equal intervals of width h , and the function is being evaluated $N + 1$ times. But there's really no reason to *require* that the intervals be of equal size. We chose this equidistant spacing while developing the primitive integration formulas so as to keep the derived expressions simple — in fact, we can obtain much more accurate formulas if we relax this requirement. We begin as we did before, trying to approximate the integral by a quadrature of the form

$$\int_a^b f(x) dx \approx \sum_{m=1}^N W_m f(x_m). \quad (4.70)$$

Note that we are starting the sum at $m = 1$, so that N refers to the number of function evaluations being made. As with the primitive integration quadratures, W_m is an unknown; now, however, the x_m are also unknown! This gives us $2N$ unknowns, so that we need $2N$ equations in order to determine them. And, as before, we obtain these equations by *requiring* that the quadrature be exact for $f(x)$ being the lowest order polynomials, $f(x) = 1, x, x^2, \dots, x^{2N-1}$. Unfortunately, the equations thus obtained are nonlinear and extremely difficult to solve (by standard algebraic methods). For example, we can take $N = 2$ and require the integration formula to be exact for $f(x) = 1, x, x^2$, and x^3 , yielding the four equations

$$(b - a) = W_1 + W_2, \quad (4.71a)$$

$$\frac{1}{2}(b^2 - a^2) = W_1 x_1 + W_2 x_2, \quad (4.71b)$$

$$\frac{1}{3}(b^3 - a^3) = W_1 x_1^2 + W_2 x_2^2, \quad (4.71c)$$

$$\frac{1}{4}(b^4 - a^4) = W_1 x_1^3 + W_2 x_2^3. \quad (4.71d)$$

Although they're nonlinear, we can solve this relatively simple set of equations analytically. The first thing is to note that any finite interval $[a, b]$ can be mapped onto the interval $[-1, 1]$ by a simple change of variable:

$$y = -1 + 2 \frac{x - a}{b - a}. \quad (4.72)$$

Thus it is sufficient for us to consider only this *normalized* integration region, in terms of which the nonlinear equations can be expressed as

$$2 = W_1 + W_2, \quad (4.73a)$$

$$0 = W_1 x_1 + W_2 x_2, \quad (4.73b)$$

$$2/3 = W_1 x_1^2 + W_2 x_2^2, \quad (4.73c)$$

$$0 = W_1 x_1^3 + W_2 x_2^3. \quad (4.73d)$$

Equations (4.73b) and (4.73d) can be combined to yield $x_1^2 = x_2^2$; since the points must be distinct (else we wouldn't have 4 independent variables), we have that $x_1 = -x_2$. Then from Equation (4.73b) we have $W_1 = W_2$, and from Equation (4.73a) we have $W_1 = 1$. Equation (4.73c) then gives us

$$2/3 = 2x_1^2,$$

or that $x_1 = 1/\sqrt{3}$. To summarize, we find that the function should be evaluated at $x_m = \pm 1/\sqrt{3}$, and that the evaluations have an equal weighting of 1. If you like, you may try to find the weights and abscissas for $N = 3$, but be forewarned: it gets harder. (*Much* harder!) This is where Professor Gauss steps in to save the day. But first, we need to know about ...

Orthogonal Polynomials

Orthogonal polynomials are one of those “advanced” topics that many of us never quite get to, although they really aren’t that difficult to understand. The basic idea is that there exists a set of polynomials $\phi_m(x)$ such that

$$\int_a^b w(x) \phi_m(x) \phi_n(x) dx = \delta_{mn} C_m. \quad (4.74)$$

(Actually, we can *construct* such a set, if we don’t already have one!) This *orthogonality condition* that orthogonal polynomials obey is simply a mathematical statement that the functions are fundamentally different. Equation (4.74) gives us the way to measure the “sameness” of two functions — if $m \neq n$, then the functions are not the same, and the integral is zero. Let’s see what this means in terms of a particular example.

The polynomials we’re most comfortable with are the ones x^0, x^1, x^2 , and so on. Start with these, and call them $u_m = x^m$. For the moment, let’s forget the weighting function, setting $w(x) = 1$, and let $a = -1$ and $b = 1$. The very first integral to think about is

$$\int_{-1}^1 \phi_0(x) \phi_0(x) dx = C_0. \quad (4.75)$$

We could simply choose $\phi_0(x) = u_0(x) = 1$, and satisfy this equation with $C_0 = 2$. Although not *necessary*, it’s often convenient to normalize these functions as well, requiring all the $C_m = 1$, so that we actually choose $\phi_0(x) = 1/\sqrt{2}$. The integral of Equation (4.74) will then yield 1, expressing the fact that the functions are identical.

The next integral to consider is

$$\int_{-1}^1 \phi_0(x) \phi_1(x) dx = 0. \quad (4.76)$$

Since the subscripts differ, this integral is required to be zero. In this case, we can choose $\phi_1(x) = u_1$ and find that Equation (4.76) is satisfied. But in general we can't count on being so lucky. What we need is a universal method for choosing ϕ_m , independent of $w(x)$, a , and b , that can be applied to whatever particular case is at hand.

Let's take ϕ_1 to be a linear combination of what we have, u_1 and ϕ_0 , and *make* it satisfy Equation (4.76). That is, we choose

$$\phi_1(x) = u_1 + \alpha_{10}\phi_0, \quad (4.77)$$

and require α_{10} to take on whatever value necessary to *force* the integral to vanish. With this expression for ϕ_1 , we have

$$\begin{aligned} \int_{-1}^1 \phi_0(x)\phi_1(x) dx &= \int_{-1}^1 \phi_0(x) [u_1 + \alpha_{10}\phi_0(x)] dx \\ &= \int_{-1}^1 \frac{1}{\sqrt{2}} \left[x + \alpha_{10}/\sqrt{2} \right] dx \\ &= 0 + \alpha_{10}. \end{aligned} \quad (4.78)$$

Since the integral is zero, we have $\alpha_{10} = 0$ and hence $\phi_1 = x$. Again, we'll normalize this result by considering the integral

$$\int_{-1}^1 \phi_1(x)\phi_1(x) dx = \int_{-1}^1 x^2 dx = \frac{2}{3}, \quad (4.79)$$

so that the normalized polynomial is

$$\phi_1(x) = \sqrt{\frac{3}{2}}x. \quad (4.80)$$

The next orthogonal polynomial is found by choosing ϕ_2 to be a linear combination of u_2 , ϕ_0 , and ϕ_1 :

$$\begin{aligned} \phi_2(x) &= u_2 + \alpha_{21}\phi_1(x) + \alpha_{20}\phi_0(x) \\ &= x^2 + \alpha_{21}\sqrt{3/2}x + \alpha_{20}/\sqrt{2}, \end{aligned} \quad (4.81)$$

and requiring *both*

$$\int_{-1}^1 \phi_0(x)\phi_2(x) dx = 0 \quad (4.82)$$

and

$$\int_{-1}^1 \phi_1(x) \phi_2(x) dx = 0. \quad (4.83)$$

From the first we find $\alpha_{20} = -\sqrt{2}/3$, and from the second $\alpha_{21} = 0$. After normalization, we find

$$\phi_2(x) = \sqrt{\frac{5}{2}} \frac{3x^2 - 1}{2}. \quad (4.84)$$

The astute reader will recognize these as the first three Legendre polynomials, although the unnormalized versions are more popular than these normalized ones. This process, known as *Gram–Schmidt orthogonalization*, can be used to generate various sets of orthogonal polynomials, depending upon $w(x)$, a , and b . (See Table 3.2.)

EXERCISE 4.12

Using this Gram–Schmidt orthogonalization process, find the next Legendre polynomial, $\phi_3(x)$.

Gaussian Integration

We now return to the evaluation of integrals. We'll consider a slightly larger class of integrals than previously indicated, by considering integrals of the form

$$\int_a^b f(x) w(x) dx = \sum_{m=1}^N W_m f(x_m), \quad (4.85)$$

where $w(x)$ is a positive definite (i.e., never negative) weighting function. (And yes, it's the same weighting function we just discussed.) Since both the weights and abscissas are treated as unknowns, we have $2N$ coefficients to be determined. Our plan will be to *require* that this quadrature be *exact* for polynomials of order $2N - 1$ and less, and use this requirement to *determine* the weights and abscissas of the quadrature!

Let $f(x)$ be a polynomial of degree $2N - 1$ (or less), and $\phi_N(x)$ be a specific orthogonal function of order N . In particular, we let ϕ_N be the orthogonal polynomial appropriate for the particular weighting function $w(x)$ and limits of integration a and b , so that

$$\int_a^b w(x) \phi_m(x) \phi_n(x) dx = \delta_{mn} C_m. \quad (4.74)$$

That is, the particular set of orthogonal polynomials to be used is dictated by the integral to be evaluated. (That's not so surprising, is it?)

Now, consider what happens if $f(x)$ is divided by $\phi_N(x)$: the leading term in the quotient will be of order $N - 1$, and the leading term in the remainder will also be of order $N - 1$. (This is not necessarily obvious, so do the division on an example of your own choosing. The division will probably not come out evenly — the remainder is what you need to *subtract* from $f(x)$ to make it come out even.) In terms of the quotient and the remainder, we can write

$$f(x) = q_{N-1}(x) \phi_N(x) + r_{N-1}(x), \quad (4.86)$$

where both $q_{N-1}(x)$ and $r_{N-1}(x)$ are polynomials of order $N - 1$.

With this expression for $f(x)$, the integral of Equation (4.85) becomes

$$\int_a^b f(x) w(x) dx = \int_a^b q_{N-1}(x) \phi_N(x) w(x) dx + \int_a^b r_{N-1}(x) w(x) dx. \quad (4.87)$$

Since the functions $\{\phi_m\}$ are a complete set, we can expand the function $q_{N-1}(x)$ as

$$q_{N-1}(x) = \sum_{i=0}^{N-1} q_i \phi_i(x), \quad (4.88)$$

where the q_i are constants. Note that the summation ranges up to $N - 1$, since $q_{N-1}(x)$ is an $(N - 1)$ -th order polynomial. The first integral on the right side of Equation (4.87) is then

$$\begin{aligned} \int_a^b q_{N-1}(x) \phi_N(x) w(x) dx &= \sum_{i=0}^{N-1} q_i \int_a^b \phi_i(x) \phi_N(x) w(x) dx \\ &= \sum_{i=0}^{N-1} q_i \delta_{iN} C_N \\ &= 0. \end{aligned} \quad (4.89)$$

Note that the integrals on the right side are merely the orthogonality integrals of Equation (4.74) and that the total integral is zero since the summation only ranges up to $N - 1$.

We began by requiring that the integration formula be exact for $f(x)$, an arbitrary function of order $2N - 1$. The product $q_{N-1}(x)\phi_N$ is also a poly-

nomial of order $2N - 1$, so it must be true that

$$\int_a^b q_{N-1}(x) \phi_N(x) w(x) dx = \sum_{m=1}^N W_m q_{N-1}(x_m) \phi_N(x_m). \quad (4.90)$$

But from Equation (4.89) we know this integral to be zero. Since $f(x)$ is arbitrary, the derived $q_{N-1}(x)$ must also be arbitrary, and so the sum doesn't vanish because of any unique characteristics of the function q_{N-1} . The only way to *guarantee* that this sum will be zero is to require that *all* the $\phi_N(x_m)$ be zero! Now, that's not as difficult as it might seem — the x_m are *chosen* such that the orthogonal polynomial $\phi_N(x)$ is zero at these points; since we need N of these x_m , we're fortunate that $\phi_N(x)$ just happens to be an N -th order polynomial and so possesses N roots. Do *you* believe in coincidence? (In a general sense, these roots might be complex. In cases of practical interest they are always real.) We've thus determined the abscissas x_m .

The integration formula is to be exact for polynomials of order $2N - 1$, so surely it must be exact for a function of lesser order as well. In particular, it must be true for the $(N - 1)$ -th order polynomial $l_{i,N}(x)$, defined as

$$l_{i,N}(x) = \frac{(x - x_1) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_N)}{(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_N)}. \quad (4.91)$$

This function occurs in connection with Lagrange's interpolating polynomial, and has the interesting property, easily verified, that

$$l_{i,N}(x_j) = \begin{cases} 0, & j \neq i \\ 1, & j = i. \end{cases} \quad (4.92)$$

We thus have the exact result

$$\int_a^b l_{i,N}(x) w(x) dx = \sum_{m=1}^N W_m l_{i,N}(x_m) = W_i, \quad (4.93)$$

so that the weights W_i can be obtained by analytically performing the indicated integration.

Let's consider an example of how this works. In particular, let's develop an integration rule for the integral

$$I = \int_{-1}^1 f(x) dx, \quad (4.94)$$

using two function evaluations. Since the limits of the integration are $a = -1$, $b = 1$, and the weighting function is $w(x) = 1$, Legendre functions are the appropriate orthogonal polynomials to use. We already know that

$$\phi_2(x) = \sqrt{\frac{5}{2}} \frac{3x^2 - 1}{2}. \quad (4.95)$$

The abscissas for the *Gauss–Legendre* integration are the zeros of this function,

$$x_1 = -\sqrt{\frac{1}{3}} \quad \text{and} \quad x_2 = +\sqrt{\frac{1}{3}}. \quad (4.96)$$

The weights are then evaluated by performing the integral of Equation (4.93). In this case,

$$\begin{aligned} W_1 &= \int_{-1}^1 l_{1,N} dx = \int_{-1}^1 \frac{x - x_2}{x_1 - x_2} dx \\ &= \frac{1}{x_1 - x_2} \left[\frac{x^2}{2} - x_2 x \right]_{-1}^1 = \frac{-2x_2}{x_1 - x_2} = 1 \end{aligned} \quad (4.97)$$

and

$$\begin{aligned} W_2 &= \int_{-1}^1 l_{2,N} dx = \int_{-1}^1 \frac{x - x_1}{x_2 - x_1} dx \\ &= \frac{1}{x_2 - x_1} \left[\frac{x^2}{2} - x_1 x \right]_{-1}^1 = \frac{-2x_1}{x_2 - x_1} = 1. \end{aligned} \quad (4.98)$$

This agrees with our previous result, but was much easier to obtain than solving a set of nonlinear equations.

EXERCISE 4.13

Determine the weights and abscissas for the Gauss–Legendre integration rule for $N = 3$.

Fortunately, we don't have to actually do all this work every time we need to perform numerical integrations by Gaussian quadrature — since they're used so frequently, weights and abscissas for various weighting functions and limits of integration have already been tabulated. For example, weights and abscissas for the Gauss–Legendre quadrature are listed in Table 4.1. Since double precision corresponds to roughly 15 decimal digits, the weights and abscissas are given to this precision. Clearly, for highly accurate work even more precision is required — one of the standard references for

this work, *Gaussian Quadrature Formulas* by Stroud and Secrest, actually presents the data to 30 digits.

TABLE 4.1 Gauss–Legendre Quadrature: Weights and Abscissas

$$\int_{-1}^1 f(x) dx = \sum_{m=1}^N W_m f(x_m)$$

x_m	W_m
$N = 2$	
$\pm 0.57735\ 02691\ 89626$	1.00000 00000 00000
$N = 3$	
$\pm 0.77459\ 66692\ 41483$	0.55555 55555 55556
0.00000 00000 00000	0.88888 88888 88889
$N = 4$	
$\pm 0.86113\ 63115\ 94053$	0.34785 48451 37454
$\pm 0.33998\ 10435\ 84856$	0.65214 51548 62546
$N = 5$	
$\pm 0.90617\ 98459\ 38664$	0.23692 68850 56189
$\pm 0.53846\ 93101\ 05683$	0.47862 86704 99367
0.00000 00000 00000	0.56888 88888 88889
$N = 6$	
$\pm 0.93246\ 95142\ 03152$	0.17132 44923 79170
$\pm 0.66120\ 93864\ 66265$	0.36076 15730 48139
$\pm 0.23861\ 91860\ 83197$	0.46791 39345 72691
$N = 7$	
$\pm 0.94910\ 79123\ 42759$	0.12948 49661 68870
$\pm 0.74153\ 11855\ 99394$	0.27970 53914 89277
$\pm 0.40584\ 51513\ 77397$	0.38183 00505 05119
0.00000 00000 00000	0.41795 91836 73469
$N = 8$	
$\pm 0.96028\ 98564\ 97536$	0.10122 85362 90376
$\pm 0.79666\ 64774\ 13627$	0.22238 10344 53375
$\pm 0.52553\ 24099\ 16329$	0.31370 66458 77887
$\pm 0.18343\ 46424\ 95650$	0.36268 37833 78362
$N = 9$	
$\pm 0.96816\ 02395\ 07626$	0.08127 43883 61574
$\pm 0.83603\ 11073\ 26636$	0.18064 81606 94857
$\pm 0.61337\ 14327\ 00590$	0.26061 06964 02936
$\pm 0.32425\ 34234\ 03809$	0.31234 70770 40003
0.00000 00000 00000	0.33023 93550 01260

$N = 10$

$\pm 0.97390\ 65285\ 17172$	$0.06667\ 13443\ 08688$
$\pm 0.86506\ 33666\ 88985$	$0.14945\ 13491\ 50581$
$\pm 0.67940\ 95682\ 99024$	$0.21908\ 63625\ 15982$
$\pm 0.43339\ 53941\ 29247$	$0.26926\ 67193\ 09996$
$\pm 0.14887\ 43389\ 81631$	$0.29552\ 42247\ 14753$

 $N = 11$

$\pm 0.97822\ 86581\ 46057$	$0.05566\ 85671\ 16174$
$\pm 0.88706\ 25997\ 68095$	$0.12558\ 03694\ 64905$
$\pm 0.73015\ 20055\ 74049$	$0.18629\ 02109\ 27734$
$\pm 0.51909\ 61292\ 06812$	$0.23319\ 37645\ 91991$
$\pm 0.26954\ 31559\ 52345$	$0.26280\ 45445\ 10247$
$0.00000\ 00000\ 00000$	$0.27292\ 50867\ 77901$

 $N = 12$

$\pm 0.98156\ 06342\ 46719$	$0.04717\ 53363\ 86512$
$\pm 0.90411\ 72563\ 70475$	$0.10693\ 93259\ 95318$
$\pm 0.76990\ 26741\ 94305$	$0.16007\ 83285\ 43346$
$\pm 0.58731\ 79542\ 86617$	$0.20316\ 74267\ 23066$
$\pm 0.36783\ 14989\ 98180$	$0.23349\ 25365\ 38355$
$\pm 0.12523\ 34085\ 11169$	$0.24914\ 70458\ 13403$

This need for precision in expressing the weights and abscissas of the quadrature introduces an additional complicating factor — how to insure that the data are entered in the program correctly, in the first place. The data can be entered in a DATA statement, or preferably, in a PARAMETER statement — something like

```
* A first attempt at entering the WEIGHTS and ABSCISSAS
* for 5-point Gauss-Legendre quadrature:
*
  Double Precision X(5),W(5)
  PARAMETER(
+ x(1)=-0.9061798459386640d0,w(1)=0.2369268550561891d0,
+ x(2)=-0.5384963101056831d0,w(2)=0.4786286704993665d0,
+ x(3)= 0.0d0,w(3)=0.5688888888888889d0,
+ x(4)= 0.5384693101056831d0,w(4)=0.4786286704993665d0,
+ x(5)= 0.9061798459386640d0,w(5)=0.2369268550561891d0)
*
```

This coding is not necessarily the most economical use of space, but that's not the idea! By using a spacing like this the numbers are relatively easy to read, and many "typo" errors will be immediately caught because the columns will

be out of alignment. But there still might be an error — a digit can simply be mistyped. But there's an easy way to check: this integration formula should be *exact*, i.e., accurate to about 15 digits, for polynomials through x^9 , so do the integral! (You were going to write an integration routine anyway, weren't you?)

EXERCISE 4.14

Evaluate the integral $\int_{-1}^1 x^m dx$, $m = 0, 1, \dots, 9$. If the weights and abscissas have been entered correctly, the results should be accurate to about 15 digits. By the way, *there are errors* in the PARAMETER statement in the listed code fragment.

This sort of verification is not particularly exciting, but it saves an immense amount of wasted effort in the long run. Assuming that you now have a valid table entered into the program, let's check the statements made concerning the accuracy of the integration.

EXERCISE 4.15

Evaluate the integral

$$\int_0^1 x^7 dx$$

using quadratures with $N = 2, 3, 4$, and 5 . Don't forget to map the given integration interval into the region $[-1, 1]$. You should observe that the error decreases as more points are used in the integration, and that the integral is obtained exactly if four or more points are used.

But polynomials are easy to evaluate!

EXERCISE 4.16

Using quadratures with 2, 3, 4, and 5 points, evaluate the integral

$$\int_0^1 e^{-x^2} dx.$$

Since the integrand is not a polynomial, the numerical evaluation is not exact. As the number of points in the quadrature is increased, however, the result becomes more and more accurate.

Composite Rules

It should be clear from the preceding discussion and from these numerical experiments that Gaussian quadrature will always be better than Romberg integration. However, going to larger quadratures is not necessarily the best thing to do. As was discussed with Romberg integration, the use of higher order polynomial approximations to the integrand is not a guarantee of obtaining a more accurate result. The alternative is to use a composite rule, in which the total integration region is divided into segments, and to use a relatively simple integration rule on each segment.

An additional advantage of a composite rule, of course, is that the error, as determined by difference in succeeding approximations, can be monitored and the integration continued until a predetermined level of accuracy has been achieved.

EXERCISE 4.17

Evaluate the integral

$$\int_0^1 e^{-x^2} dx,$$

by a composite rule, using 4-point Gauss–Legendre integration within each segment. For comparison, repeat the evaluation using the trapezoid rule and Romberg integration. Compare the effort needed to obtain 8-significant-figure accuracy with these different methods.

Gauss–Laguerre Quadrature

To use Gauss–Legendre quadrature the limits of the integration must be finite. But it can happen, and often does, that the integral of interest extends to infinity. That is, a physically significant quantity might be expressed by the integral

$$I = \int_0^{\infty} g(x) dx. \quad (4.99)$$

For this integral to exist, $g(x)$ must go to zero asymptotically, faster than $1/x$. It might even happen that the particular integral of interest is of the form

$$I = \int_0^{\infty} e^{-x} f(x) dx. \quad (4.100)$$

In this case, $f(x)$ can be a polynomial, for example, since the factor e^{-x} will cause the integrand to vanish asymptotically.

In order to develop a Gauss-style integration formula, we need a set of functions that are orthogonal over the region $[0, \infty]$ with the weighting function $w(x) = e^{-x}$. Proceeding as before, we can *construct* a set of polynomials that has precisely this characteristic! Beginning (again) with the set $u_m = x^m$, we first consider the function $\phi_0 = \alpha_{00}u_0$, and the integral

$$\int_0^\infty w(x) \phi_0(x) \phi_0(x) dx = \alpha_{00}^2 \int_0^\infty e^{-x} dx = \alpha_{00}^2 = C_0. \quad (4.101)$$

With C_0 set to unity, we find $\phi_0(x) = 1$. We then consider the next polynomial,

$$\phi_1(x) = u_1(x) + \alpha_{10}\phi_0(x), \quad (4.102)$$

and require that

$$\int_0^\infty e^{-x} \phi_0(x) \phi_1(x) dx = 0, \quad (4.103)$$

and so on. This process constructs the *Laguerre* polynomials; the zeros of these functions can be found, the appropriate weights for the integration determined. These can then be tabulated, as in Table 4.2. We have thus found the sought-after Gauss–Laguerre integration formulas,

$$\int_0^\infty e^{-x} f(x) dx = \sum_{m=1}^N W_m f(x_m). \quad (4.104)$$

TABLE 4.2 Gauss–Laguerre Quadrature: Weights and Abscissas

$\int_0^\infty e^{-x} f(x) dx = \sum_{m=1}^N W_m f(x_m)$	
x_m	W_m
$N = 2$	
5.85786 43762 69050(-1)	8.53553 39059 32738(-1)
3.41421 35623 73095	1.46446 60940 67262(-1)
$N = 4$	
3.22547 68961 93923(-1)	6.03154 10434 16336(-1)
1.74576 11011 58347	3.57418 69243 77997(-1)
4.53662 02969 21128	3.88879 08515 00538(-2)
9.39507 09123 01133	5.39294 70556 13275(-4)
$N = 6$	
2.22846 60417 92607(-1)	4.58964 67394 99636(-1)

1.18893 21016 72623	4.17000 83077 21210(-1)
2.99273 63260 59314	1.13373 38207 40450(-1)
5.77514 35691 04511	1.03991 97453 14907(-2)
9.83746 74183 82590	2.61017 20281 49321(-4)
1.59828 73980 60170(1)	8.98547 90642 96212(-7)

 $N = 8$

1.70279 63230 51010(-1)	3.69188 58934 16375(-1)
9.03701 77679 93799(-1)	4.18786 78081 43430(-1)
2.25108 66298 66131	1.75794 98663 71718(-1)
4.26670 01702 87659	3.33434 92261 21565(-2)
7.04590 54023 93466	2.79453 62352 25673(-3)
1.07585 16010 18100(1)	9.07650 87733 58213(-5)
1.57406 78641 27800(1)	8.48574 67162 72532(-7)
2.28631 31736 88926(1)	1.04800 11748 71510(-9)

 $N = 10$

1.37793 47054 04924(-1)	3.08441 11576 50201(-1)
7.29454 54950 31705(-1)	4.01119 92915 52736(-1)
1.80834 29017 40316	2.18068 28761 18094(-1)
3.40143 36978 54900	6.20874 56098 67775(-2)
5.55249 61400 63804	9.50150 69751 81101(-3)
8.33015 27467 64497	7.53008 38858 75388(-4)
1.18437 85837 90007(1)	2.82592 33495 99566(-5)
1.62792 57831 37810(1)	4.24931 39849 62686(-7)
2.19965 85811 98076(1)	1.04800 11748 71510(-9)
2.99206 97012 27389(1)	9.91182 72196 09009(-12)

 $N = 12$

1.15722 11735 80207(-1)	2.64731 37105 54432(-1)
6.11757 48451 51307(-1)	3.77759 27587 31380(-1)
1.51261 02697 76419	2.44082 01131 98776(-1)
2.83375 13377 43507	9.04492 22211 68093(-2)
4.59922 76394 18348	2.01012 81154 63410(-2)
6.84452 54531 15177	2.66397 35418 65216(-3)
9.62131 68424 56867	2.03231 59266 29994(-4)
1.30060 54993 30635(1)	8.36505 58568 19799(-6)
1.71168 55187 46226(1)	1.66849 38765 40910(-7)
2.21510 90379 39701(1)	1.34239 10305 15004(-9)
2.84879 67250 98400(1)	3.06160 16350 35021(-12)
3.70991 21044 46692(1)	8.14807 74674 26242(-16)

 $N = 18$

7.81691 66669 70547(-2)	1.85588 60314 69188(-1)
4.12490 08525 91293(-1)	3.10181 76637 02253(-1)
1.01652 01796 23540	2.67866 56714 85364(-1)
1.89488 85099 69761	1.52979 74746 80749(-1)
3.05435 31132 02660	6.14349 17860 96165(-2)
4.50420 55388 89893	1.76872 13080 77293(-2)
6.25672 50739 49111	3.66017 97677 59918(-3)
8.32782 51566 05630	5.40622 78700 77353(-4)
1.07379 90047 75761(1)	5.61696 50512 14231(-5)
1.35136 56207 55509(1)	4.01530 78837 01158(-6)
1.66893 06281 93011(1)	1.91466 98566 75675(-7)
2.03107 67626 26774(1)	5.83609 52686 31594(-9)

2.44406 81359 28370(1)	1.07171 12669 55390(-10)
2.91682 08662 57962(1)	1.08909 87138 88834(-12)
3.46279 27065 66017(1)	5.38666 47483 78309(-15)
4.10418 16772 80876(1)	1.04986 59780 35703(-17)
4.88339 22716 08652(1)	5.40539 84516 31054(-21)
5.90905 46435 90125(1)	2.69165 32692 01029(-25)

Since the upper limit of the integration is infinite, we cannot develop composite formulas with Gauss–Laguerre integration. Of course, we can use Gauss–Legendre integration, using composite formulas if needed, up to some (arbitrary) finite limit, and then use Gauss–Laguerre integration from that finite limit up to infinity (with an appropriate change of variable, of course).

EXERCISE 4.18

The integral

$$\int_0^{\infty} \frac{x^3}{e^x - 1} dx$$

appears in Planck’s treatment of black body radiation. Perform the integral numerically, with Gauss–Laguerre integration using 2, 4, 6, and 8 points.

Multidimensional Numerical Integration

Now that we can integrate in one dimension, it might seem that the jump to two or more dimensions would not be particularly difficult. But actually it is! First is the issue of the number of function evaluations: if a typical integration requires, say, 100 points in one dimension, then a typical integration in two dimensions should take 10,000 points, and in three dimensions a million points are required. That’s a lot of function evaluations! A second difficulty is that instead of simply specifying the limits of integration, as is done in one dimension, in two dimensions a *region* of integration is specified, perhaps as bounded by a particular curve. And in three dimensions it’s a volume, as specified by a boundary surface — so simply specifying the region of integration can become difficult. In general, multidimensional integration is *a lot harder* than integration in a single variable.

Let’s imagine that we need to evaluate the integral

$$I = \int_a^b \int_c^d f(x, y) dx dy. \quad (4.105)$$

If a , b , c , and d are constants, then the integration region is simply a rectangle in the xy -plane, and the integral is nothing more than

$$I = \int_a^b F(y) dy, \quad (4.106)$$

where

$$F(y) = \int_c^d f(x, y) dx. \quad (4.107)$$

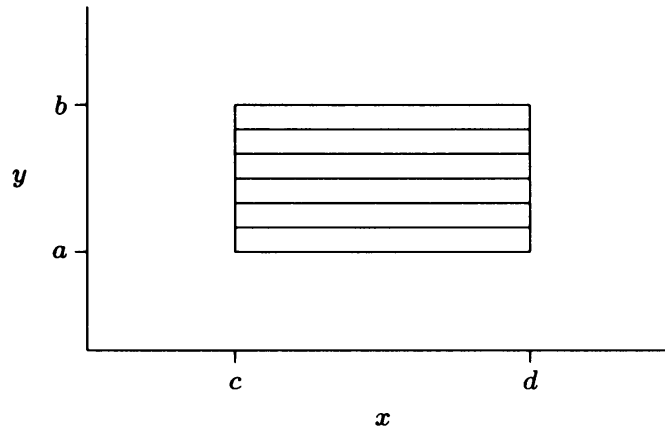


FIGURE 4.4 Two-dimensional integration, doing the x -integral “first.”

Figure 4.4 suggests what’s happening — the rectangular integration region is broken into strips running in the x -direction: the area under the function $f(x, y)$ along each strip is obtained by integrating in x , and the total area is obtained by adding the contributions from all the strips. Of course, we could equally well have chosen to perform the y integral first, so that

$$I = \int_c^d G(x) dx, \quad (4.108)$$

where

$$G(x) = \int_a^b f(x, y) dy. \quad (4.109)$$

As indicated in Figure 4.5, this corresponds to running the strips along the y -axis.

The computer code to perform such two-dimensional integrals is easily obtained from the one-dimensional code. Although a single subroutine with nested loops would certainly do the job, it might be easier simply to have two subroutines, one that does the x -integral and one that does the y -integral.

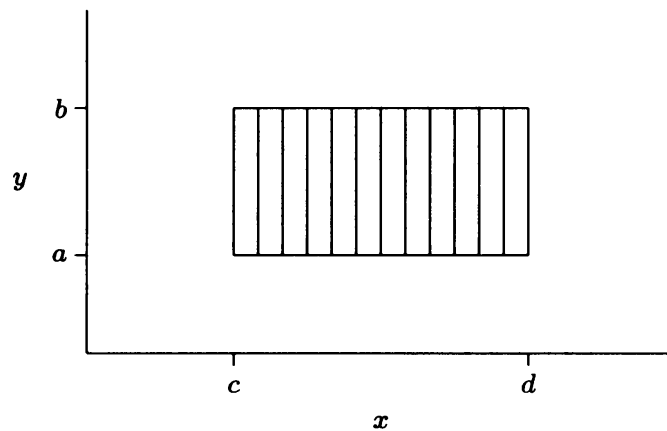


FIGURE 4.5 Two-dimensional integration, doing the y -integral “first.”

Of course, what goes into each will depend upon the order in which you choose to perform the integrations. Let’s say that you’ve decided to do the x -integral “first,” so that the integral is to be evaluated according to Equations (4.106) and (4.107), and as depicted in Figure 4.4. Then the code might look something like this:

```

* The next code does two-dimensional numerical integration,
* using the function FofY to perform the integration in
* the x-dimension.
*
...
* Do all necessary setup...
....
* The DO loop does the y-integral, Equation (4.106).
*
    total = 0.d0
    DO i = 1, n
        ...
        y = ...
        total = total + W(i) * FofY( c, d, y )
    END DO
    ...
end

*
*-----
*
      Double Precision Function FofY( c, d, y )
*
* This subroutine evaluates the integral of f(x,y) dx
* between the limits of x=c and x=d, i.e., F(y) of

```

```

* Equation (4.107). With regard to this x-integration,
* y is a constant.
*
      double precision ...
      ...
*
* This DO loop does the x-integral.
*
      FofY = 0.d0
      DO i=1,nn
          ...
          x = ...
          FofY = FofY + ww(i) * f(x,y)
      END DO
      ...
      end

```

This is only an outline of the appropriate computer code — the method of integration (trapezoid, Simpson, Gaussian, etc.) hasn't even been specified. And, of course, whatever method used should be put together so as to guarantee that either convergence has been obtained or an appropriate message will be displayed. But these are problems that you've already encountered.

EXERCISE 4.19

Numerically integrate

$$\iint e^{-xy} dx dy$$

on the rectangular domain defined by $0 \leq x \leq 2$ and $0 \leq y \leq 1$. Use your personal preference for method of integration, but be sure that your result is accurate to 8 significant figures.

Other Integration Domains

Integration on a rectangular domain is a rather straightforward task, but we often need to perform the integration over different regions of interest. For example, perhaps we need the integral evaluated in the first quadrant, bounded by a circle of radius 1, as indicated in Figure 4.6. The y -variable still varies from 0 to 1, but the limits of the x -integration vary from 0 to $\sqrt{1-y^2}$. In general, the limits of the integration are a function of all variables yet to be integrated. The computer code just discussed is still appropriate for this problem except that d , the upper limit of the x -integration, is not a constant and is

different for each evaluation of the function $F \circ fY$, i.e., for each x -integration. The function $F \circ fY$ itself would remain unchanged.

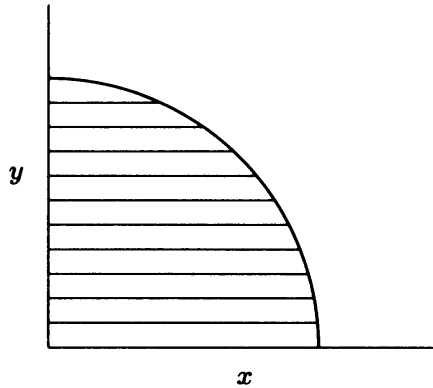


FIGURE 4.6 A two-dimensional integration region divided into Cartesian strips.

EXERCISE 4.20

Modify your program so that the calling routine adjusts the limits of the integration. (No changes are needed in the *called* function, however.) Then evaluate the integral

$$I = \int_0^1 \left(\int_0^{\sqrt{1-y^2}} e^{-xy} dx \right) dy$$

over the quarter circle of unit radius lying in the first quadrant.

It should be noted that a change of variables can sometimes be used to simplify the specification of the bounding region. For example, if the integration region is bounded by a circle, then it might be advantageous to change from cartesian to polar coordinates. This would yield integration domains as indicated in Figure 4.7.

EXERCISE 4.21

Evaluate the integral

$$I = \int_0^1 \left(\int_0^{\sqrt{1-y^2}} e^{-xy} dx \right) dy$$

over the quarter circle of radius 1 lying in the first quadrant, by first

changing to polar coordinates. Note that in these coordinates, the limits of integration are constant.

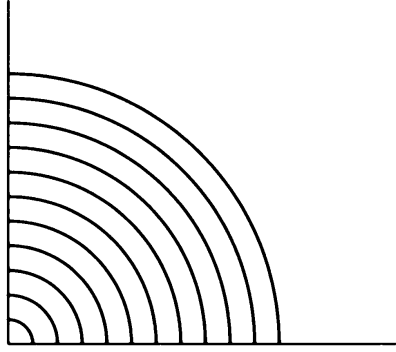


FIGURE 4.7 A two-dimensional integration region divided into polar strips.

A Little Physics Problem

Consider a square region in the xy -plane, such that $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$, containing a uniform charge distribution ρ , as depicted in Figure 4.8. The electrostatic potential at the point (x_p, y_p) due to this charge distribution is obtained by integrating over the charged region,

$$\Phi(x_p, y_p) = \frac{\rho}{4\pi\epsilon_0} \int_{-1}^1 \int_{-1}^1 \frac{dx dy}{\sqrt{(x - x_p)^2 + (y - y_p)^2}}. \quad (4.110)$$

For simplicity, take $\rho/4\pi\epsilon_0$ to be 1.

EXERCISE 4.22

Use your two dimensional integration routine to evaluate $\Phi(x_p, y_p)$, and create a table of values for $x_p, y_p = 2, 4, \dots, 20$. Use a sufficient number of points in your integration scheme to guarantee 8-significant-digit accuracy in your final results.

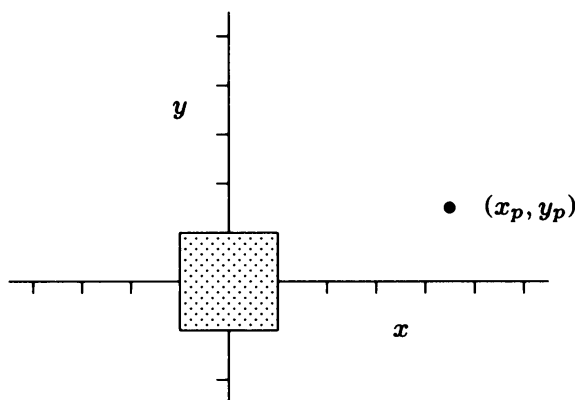


FIGURE 4.8 A uniformly charged square region in Cartesian coordinates.

More on Orthogonal Polynomials

One reason that orthogonal functions in general, and Legendre functions in particular, are important is that they allow us to write a complicated thing, such as the electrostatic potential of a charged object, in terms of just a few coefficients. That is, the potential might be written as

$$\Phi(r, \theta) = \sum_{i=0}^{\infty} a_i(r) P_i(\cos \theta). \quad (4.111)$$

A relatively few coefficients are then sufficient to describe the potential to high accuracy, rather than requiring a table of values. Note that the coefficients $a_i(r)$ are functions of r , but are *constants with respect to* θ . Factoring the potential in this way divides the problem into two portions: the angular portion, which is often geometric in nature and easily solved, and the radial portion, which is where the real difficulty of the problem often resides.

The Legendre functions used here are the usual, *unnormalized* ones, the first few of which are

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_2(x) &= (3x^2 - 1)/2 \\ P_3(x) &= (5x^3 - 3x)/2 \\ P_4(x) &= (35x^4 - 30x^2 + 3)/8 \\ P_5(x) &= (63x^5 - 70x^3 + 15x)/8 \end{aligned} \quad (4.112)$$

The orthogonality condition for these unnormalized functions is

$$\int_0^\pi P_m(\cos \theta) P_n(\cos \theta) \sin \theta d\theta = \frac{2}{2m+1} \delta_{mn}. \quad (4.113)$$

Equation (4.111) can then be multiplied by $P_j(\cos \theta)$ and integrated to yield

$$a_j(r) = \frac{2j+1}{2} \int_0^\pi \Phi(r, \theta) P_j(\cos \theta) \sin \theta d\theta. \quad (4.114)$$

Due to the nature of the integrand, Gaussian quadrature is well suited to evaluate this integral, although a change of variables is necessary since the specified integration is on the interval $[0, \pi]$.

Let's reconsider the problem of the uniformly charged square. If you have not already done so, you should write a subroutine (or function) that performs the two-dimensional integral of Equation (4.110), returning the value of the potential at the point (x_p, y_p) . This function evaluation will be needed to evaluate the integral for $a_i(r_p)$, as given in Equation (4.114). Of course, the points at which the function is to be evaluated are determined by r_p and the abscissas at which the θ_p integral is to be performed. That is, measuring θ counterclockwise from the positive x -axis, we have

$$\begin{aligned} x_p &= r_p \cos \theta_p \\ \text{and } y_p &= r_p \sin \theta_p, \end{aligned} \quad (4.115)$$

as illustrated in Figure 4.9.

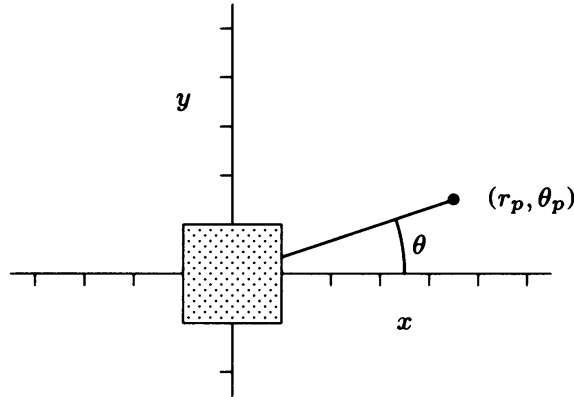


FIGURE 4.9 A uniformly charged square region in polar coordinates.

EXERCISE 4.23

Expand the potential in a series of Legendre functions through P_5 . That is, evaluate the r_p -dependent coefficients $a_i(r_p)$ for $i = 0, \dots, 5$ at $r_p = 2, 4, \dots, 20$. Perform the required integrations by Gaussian quadrature, using an appropriate number of points N in the integration. *Justify your choice of N .*

Monte Carlo Integration

The integration methods discussed so far all are based upon making a polynomial approximation to the integrand. But there are other ways of calculating an integral, some of which are *very* different. One class of these methods relies upon random numbers; these methods have come to be known under the general rubric *Monte Carlo*, after the famous casino.

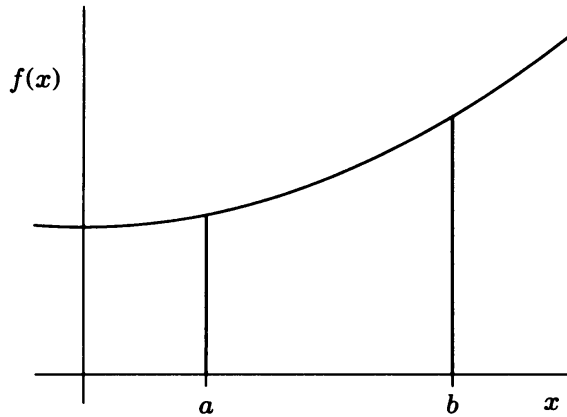


FIGURE 4.10 The integral of $f(x)$ between a and b .

Consider a function to be integrated, as in Figure 4.10 — the integral is just the area under the curve. If we knew the area, we could divide by the width of the interval, $(b - a)$, and define the *average* value of the function, $\langle f \rangle$. Conversely, the width times the average value of the function is the integral,

$$\int_a^b f(x) dx = (b - a) \langle f \rangle. \quad (4.116)$$

So if we just had some way of calculating the average ...

And that's where the need for random numbers arises. Let's imagine that we have a "list" of random numbers, the x_i , uniformly distributed be-

tween a and b . To calculate the function average, we simply evaluate $f(x)$ at each of the randomly selected points, and divide by the number of points:

$$\langle f \rangle_N = \frac{1}{N} \sum_{i=1}^N f(x_i). \quad (4.117)$$

As the number of points used in calculating the average increases, $\langle f \rangle_N$ tends towards the “real” average, $\langle f \rangle$, and so we write the Monte Carlo estimate of the integral as

$$\int_a^b f(x) dx \approx (b-a) \frac{1}{N} \sum_{i=1}^N f(x_i). \quad (4.118)$$

Finding a “list” of random numbers could be a real problem, of course. Fortunately for us, it’s a problem that has already been tackled by others, and as a result *random number generators* are fairly common. Unfortunately, there is a wide range in the quality of these generators, with some of them being quite unacceptable. For a number of years a rather poor random number generator was widely distributed in the scientific community, and more recently algorithms now described as “mediocre” were circulated. For our purposes, which are not particularly demanding, the subroutine RANDOM, supplied with the FORTRAN compiler, will suffice. If our needs became more stringent, however, verifying the quality of the generator, and replacing it if warranted, would take a high priority. In any event, we should note that these numbers are generated by a computer algorithm, and hence are not truly random — they are in fact *pseudo-random* — but they’ll serve our purpose. Unfortunately, the argument of RANDOM must be a REAL variable; we also need the subroutine SEED, which initializes the random number generator. A suitable code for estimating the integral then would be

```

      Program MONTE_CARLO
*
*   Paul L. DeVries, Department of Physics, Miami University
*
*   This program computes a Monte Carlo style estimate of
*   the integral exp(x) between 0 and 1.  (= e-1)
*
      double precision sum,e,ran1, x, error, monte
      real xxx
      integer N,i
      integer*2 value
      parameter ( e = 2.718281828459045d0 )
*

```

```

* Initialize the "seed" used in the Random Number
* Generator, and set the accumulated SUM to zero.
*
      value = 1
      call seed( value )
      sum =0.d0
*
* Calculate the function a total of 1,000 times, printing
* an estimate of the integral after every 10 evaluations.
*
      DO i = 1, 100
*
* Evaluate the function another 10 times.      SUM is the
* accumulated total.
*
          DO j = 1, 10
              call random( xxx )
              x = xxx
              sum = sum + exp(x)
          END DO
*
* The function has now been evaluated a total of
* ( 10 * i ) times.
*
          N = i * 10
          MONTE = sum / N
*
* Calculate the relative error, from the known value
* of the integral.
*
          error = abs( monte - (e-1.d0) )/( e - 1.d0 )
          write(*,*) n, MONTE, error
      END DO

      end

```

Before using RANDOM, SEED is called to initialize the generation of a sequence of random numbers. The random number generator will provide the same sequence of random numbers every time it's called, after it's been initialized with a particular value of value. This is essential to obtaining reproducible results and in debugging complicated programs. To obtain a *different* sequence of random numbers, simply call SEED with a different initial value.

This code prints the estimate of the integral after every 10 function

evaluations; typical results (which is to say, the ones I found) are presented in Figure 4.11. With 1,000 function evaluations the Monte Carlo estimate of the integral is 1.7530885, compared to $e-1 \approx 1.7182818$, for an accuracy of about two significant digits. The figure suggests that the estimate of the integral has stabilized, if not converged, to this value. This is an illusion, however; as the number of accumulated points grow, the influence of the last few points diminishes, so that the variation from one estimate to the next is necessarily reduced.

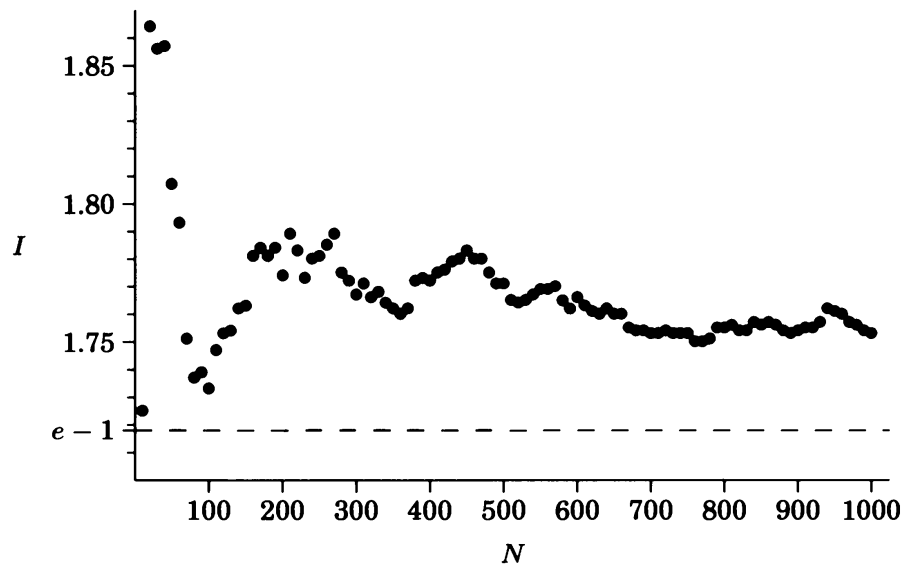


FIGURE 4.11 Monte Carlo estimates of the integral $\int_0^1 e^x dx$ using various numbers of sampling points. The correct result is indicated by the dashed line.

The accuracy of the Monte Carlo method can be enhanced by using information about the function. For example, if $g(x) \approx f(x)$, and if we can integrate g , then we can write

$$\int_a^b f(x) dx = \int_a^b \frac{f(x)}{g(x)} g(x) dx = \int_{y^{-1}(a)}^{y^{-1}(b)} \frac{f(x)}{g(x)} dy, \quad (4.119)$$

where

$$y(x) = \int_a^x g(t) dt. \quad (4.120)$$

Instead of uniformly sampling x to integrate $f(x)$, we uniformly sample y and integrate $f(x)/g(x)$! To the extent that g is a good approximation to f , the

integrand will be unity, and easy to evaluate. This technique, known as *importance sampling*, has the effect of placing a larger number of sample points where the function is large, thus yielding a better estimate of the integral.

EXERCISE 4.24

Consider the integral

$$I = \int_0^1 e^x dx. \quad (4.121)$$

Since $e^x \approx 1 + x$, the integral can be rewritten as

$$I = \int_0^1 \frac{e^x}{1+x} (1+x) dx = \int_0^{3/2} \frac{e^{\sqrt{1+2y}-1}}{\sqrt{1+2y}} dy, \quad (4.122)$$

where

$$y = \int_0^x (1+t) dt = x + \frac{x^2}{2} \quad (4.123)$$

and

$$x = -1 + \sqrt{1+2y}. \quad (4.124)$$

This change of variables modifies the limits of integration and the form of the integrand, of course. To evaluate the integral in its new form, y is to be uniformly sampled on the interval $[0, 3/2]$. Modify the previous Monte Carlo program to evaluate this integral.

You probably found a better result than I had obtained, but not much better. Particularly when you consider that with 1025 function evaluations — 1024 intervals — the composite trapezoid rule yields 1.7182824, accurate to 7 significant digits. (Simpson's rule gives this same level of accuracy with only 129 function evaluations!) So why do we care about Monte Carlo methods?

Actually, if the integral can be done by other means, then Monte Carlo *is not* a good choice. Monte Carlo methods come into their own in situations where the integral is difficult, or even impossible, to evaluate in any other way. And in order to explain the advantages of Monte Carlo methods in these cases, we need first to consider some of the ideas from probability theory.

Let's say that we have a Monte Carlo estimate of an integral, obtained with the first 100 random numbers we generated. And then we make another estimate of the integral, using the *next* 100 random numbers. Would these estimates be the same? Of course not! (Unless the integrand is a constant — a rather uninteresting case.) A different set of random numbers would in all likelihood yield a different estimate, although perhaps not too different. And as a larger and larger number of estimates are considered, we would expect a

smooth distribution of estimates to be observed — most of them near the true value of the integral, with the number of estimates decreasing as we moved away from that true value.

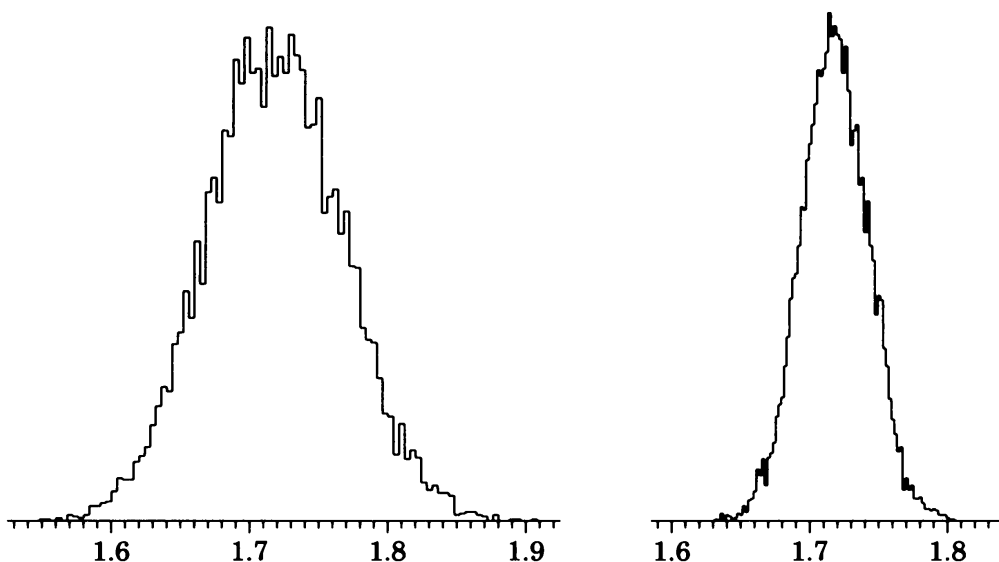


FIGURE 4.12 Distributions of 10,000 Monte Carlo estimates of the integral $\int_0^1 e^x dx$. On the left, each integral was evaluated with $N = 100$ points; on the right, with $N = 400$ points.

And that’s exactly what we see! In Figure 4.12, two distributions of 10,000 estimates of the integral are displayed. The distribution on the left was obtained by using 100 points in the estimate of the integral. The plot should look (at least vaguely) familiar to you — something like a bell shape. In fact, the Central Limit Theorem of probability theory states that, if N is sufficiently large, the distribution of sums will be a normal distribution, e.g., described by a Gaussian function.

Now, what would happen if we used more points N to estimate the integral? It would seem reasonable to expect to get a “better” answer, in the sense that if a large number of estimates of the integral were made, the distribution of estimates would be narrower about the true answer. And indeed, this is what’s seen on the right side of Figure 4.12, where another 10,000 estimates of the integral are plotted. This time, each estimate of the integral was obtained using $N = 400$ points.

(To generate these histograms, we kept track of how many Monte Carlo estimates fell within each narrow “bin.” The plot is simply the number

of estimates in each bin. For $N = 100$, the bin was 0.004 wide; for the $N = 400$ example, it was reduced to 0.002.)

Comparing the two distributions in the figure, we immediately see that the second distribution is much narrower than the first. We can take a ruler and measure the width of the distributions, say, at half their maximum values, and find that the second distribution is very nearly one-half the width of the first. This is another fundamental result from probability theory — that the width of the distribution of estimates of an integral is proportional to $1/\sqrt{N}$, so that when we quadrupled the number of points, we halved the width of the distribution.

Probability theory also tells us how to estimate the standard deviation of the mean, a measure of the width of the distribution of estimates. Since 68.3% of all estimates lie within one standard deviation of the mean, we can also say that there is a 68.3% probability that our particular estimate $\langle f \rangle_N$ lies within one standard deviation of the exact average $\langle f \rangle$! (There's also a 95.4% probability of being within two standard deviations, a 99.7% probability of being within three standard deviations, and so on.) The standard deviation can be estimated from the points sampled in evaluating the integral:

$$\sigma_N = \sqrt{\frac{\frac{1}{N} \sum f(x_i)^2 - \left(\frac{1}{N} \sum f(x_i) \right)^2}{N-1}}. \quad (4.125)$$

It's important to note that σ_N is accumulated as more points are sampled. That is, the two sums appearing in Equation (4.125) are updated with every additional point, and σ_N can be evaluated whenever it's needed or desired. Thus

$$\int_a^b f(x) dx \approx (b-a) (\langle f \rangle_N \pm \sigma_N), \quad (4.126)$$

with 68.3% confidence. Implicit in this is one of the strengths of the Monte Carlo method — if a more accurate estimate of the integral is desired, you need only to sample the integrand at more randomly selected points. As N increases, σ_N decreases, and the probability of your result lying within some specified vicinity of the correct result increases. But also contained is its weakness — the improvement goes only as the square root of N .

EXERCISE 4.25

Modify the Monte Carlo code to include the calculation of the standard

deviation. Use your code to estimate the integral

$$I = \int_0^\pi \sin x \, dx \quad (4.127)$$

and its standard deviation. In general, the addition of importance sampling will greatly increase the accuracy of the integration. Noting that

$$\sin x \approx \frac{4}{\pi^2} x (\pi - x), \quad 0 \leq x \leq \pi, \quad (4.128)$$

reevaluate the integral using importance sampling, and compare the estimated standard deviations.

The “error” in a Monte Carlo calculation is fundamentally different from that in the other methods of integration that we’ve discussed. With the trapezoid rule, for example, the error represents the inadequacy of the linear approximation in fitting the actual integrand being integrated — by making the step size smaller, the fit is made better, and the error decreases. Since the error $\sim h^2$, the error can be halved if h is decreased by $\sqrt{2}$, which is to say that N would be increased by $\sqrt{2}$. In a two-dimensional integral, the step sizes in both dimensions would have to be decreased, so that N would need to be increased by a factor of 2. In a general, multidimensional integral of dimension d , N must be increased by a factor of $2^{d/2}$ in order to decrease the error by a factor of 2.

In a Monte Carlo calculation, the “error” is of a probabilistic nature — 68.3% of the time, the estimate is within one standard deviation of the “correct” answer. As more points are included, the average gets better, in that sense. To perform a multidimensional integration, the random number generator will be called d times to get each of d coordinate values. Then the function is evaluated, and the sums updated. Although the Monte Carlo method converges slowly — only the square root of N — this convergence is dependent upon the probabilistic nature of the averaging process, and not the dimensionality of the integral! That is, to reduce the error by 2, N must be increased by 4, *independent of the dimensionality of the integral!* Thus the convergence of the Monte Carlo method is comparable to the trapezoid rule in four dimensions, and faster if $d > 4$. For integrals of sufficiently high dimensionality, Monte Carlo methods actually converge *faster* than any of the other methods we’ve discussed!

In some areas of physics, such multidimensional integrals occur frequently. In statistical mechanics, for example, a standard problem goes like this: Given a microscopic variable u which is defined at every point in phase

space, the equilibrium thermodynamic value \bar{u} is given as

$$\bar{u} = \frac{\int u \exp [-E/kT] dx^{3N} dv^{3N}}{\int \exp [-E/kT] dx^{3N} dv^{3N}}, \quad (4.129)$$

where the notation $dx^{3N} dv^{3N}$ means to integrate over the three components of position and velocity for each of N particles. N doesn't need to be very large for the integral to become intractable to all but Monte Carlo methods of attack.

And even in situations where the standard methods are applicable, Monte Carlo might still be preferred, purely on the number of function evaluations required. For example, to evaluate a 10-dimensional integral, using only 10 points per coordinate, requires 10 billion function evaluations. That might take a while. It's not unusual for symmetry considerations to reduce the complexity of the problem substantially, but still — we're not going to evaluate an integral like that by direct methods ! With Monte Carlo methods we can at least obtain *some* estimate, and a reasonable idea of the error, with however many points we have. And to improve the result, we need only to add more function evaluations to the approximation. Clearly, this process is not likely to yield a highly precise result — but it can give valid estimates of 1- or 2-significant-digit accuracy where other methods totally fail.

EXERCISE 4.26

Evaluate the 9-dimensional integral

$$I = \int_0^1 \cdots \int_0^1 \frac{da_x da_y da_z db_x db_y db_z dc_x dc_y dc_z}{(\vec{a} + \vec{b}) \cdot \vec{c}}.$$

Use a sufficient number of points in the integration so that the estimated standard deviation is less than 10% of the estimated integral.

Can you find some other way to evaluate the integral? An additional feature of Monte Carlo integration is that singularities don't bother it too much — this can't be said of other integration schemes. Integrals similar to the one in the exercise appear in the study of electron plasmas, but they are more complicated in that the integration extends over all of space. By an appropriate change of variable and the use of importance sampling, Monte Carlo methods can be used to give (crude) estimates of these integrals, where no other methods can even be applied.

Monte Carlo Simulations

In addition to evaluating multidimensional integrals, Monte Carlo methods are widely used to mimic, or simulate, “random” processes. In fact, one of the first uses of Monte Carlo methods was in designing nuclear reactors — in particular, to determine how much shielding is necessary to stop neutrons. One could imagine following the path of a neutron as it moved through the shielding material, encountering various nuclei and being scattered. But after a few such collisions, any “memory” the neutron had of its initial conditions would be lost. That is, the specific result of a particular collision would have no correlation with the initial conditions, i.e., it would be “random.” Such processes are said to be *stochastic*. Such stochastic problems, or physical problems that are treatable by stochastic methods, abound in physics. And in many cases, the use of simulations is the only practical tool for their investigation.

As a rather trivial example, consider the drunken sailor problem. A young seaman, after many months at sea, is given shore leave in a foreign port. He and his buddies explore the town, find an interesting little bistro and proceed to partake of the local brew. In excess, unfortunately. When it’s time to return to the ship, the sailor can hardly stand. As he leaves the bistro, a physicist sitting at the end of the bar observes that the sailor is equally likely to step in any direction. In his condition, how far will he travel after taking N steps?

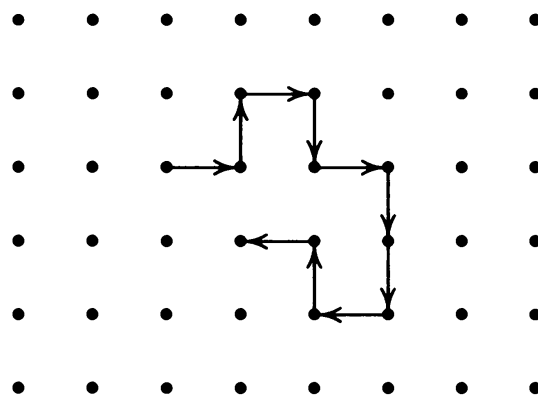


FIGURE 4.13 A “typical” random walk on a two-dimensional square grid.

We can *simulate* this walk by using random numbers. Let’s imagine a square grid, with an equal probability of moving on this grid in any of four directions. A random number generator will be called upon to generate the direction: if between 0 and 0.25, move north; between 0.25 and 0.50, move

east; and so on. A typical path is shown in Figure 4.13. Of course, we don't learn much from a single path — we need to take a large number of paths, and build up a distribution. Then we can state, in a probabilistic sense, how far the sailor is likely to travel.

EXERCISE 4.27

Write a computer code to investigate the random walk problem. Plot the mean distance traveled versus the number of steps taken. For large N , can you make a statement concerning the functional relationship between these quantities?

It might seem that this exercise is pure fancy, having little to do with physics. But what if we were investigating the mobility of an atom attached to the surface of a crystal? With the atom playing the role of the sailor, and the grid points corresponding to lattice points of the crystal, we have a description of a real physical situation. And we can use the simulation to ask *real* physical questions: How far will the atom travel in a given period of time? What percentage of the surface needs to be covered to insure that two atoms will be at adjoining sites 50% of the time? As the percentage of coverage increases, will patterns of atoms develop on the surface?

In the example, only steps taken in the cardinal directions were permitted. This restriction can be relaxed, of course, and we can also consider motion in three dimensions. For example, consider the problem of diffusion. At room temperature, molecules in the air are traveling at hundreds of meters per second. Yet, when a bottle of perfume is opened at the far end of the classroom, it takes several minutes for the aroma to be perceived at the other end. Why?

The explanation is that while the velocity of the molecules is great, there are a large number of collisions with other molecules. Each such collision changes the direction of the aromatic molecule, so that it wanders about, much like our drunken sailor, making many collisions while achieving only modest displacement from its origin. Let's model this process with our Monte Carlo approach.

We begin with a single molecule, and allow it to travel in a random direction. The first problem, then, is determining the direction. We want a uniform distribution of directions, but an element of solid angle is

$$d\Omega = \sin \theta \, d\theta \, d\phi. \quad (4.130)$$

If we were simply to take a uniform distribution in θ and in ϕ , there would be

a “bunching” of chosen directions about the poles. What we really want is a $\sin \theta$ distribution so that the directions are uniformly distributed throughout space, i.e., uniformly distributed on the unit sphere. (This is very similar to what we encountered with importance sampling — changing the variables in such a way as to put points where we want them. This similarity is not mere coincidence: ultimately we will take a large number of events and average over them, and hence *are* performing an integration.) In this case, let’s introduce the variable g such that

$$d\Omega = dg d\phi, \quad (4.131)$$

so that g and ϕ should be uniformly sampled. But that means that

$$dg = \sin \theta d\theta \quad (4.132)$$

or that

$$g(\theta) = \cos \theta. \quad (4.133)$$

To obtain our uniform distribution of solid angles, we select ϕ from a uniform distribution of random variables on the interval $[0, 2\pi]$. We then select g from a uniform distribution on $[-1, 1]$, and obtain θ from the relation

$$\theta = \cos^{-1} g. \quad (4.134)$$

After choosing a direction, we allow the molecule to move some given distance before colliding with a molecule of air. Realistically, this distance is another variable of the problem, but we’ll assume it to be a constant, taken to be the mean free path. As a result of the collision, the molecule is scattered in a random direction, travels another mean free path distance, and so on. Slowed by all these collisions, how far will the molecule travel in a given time? Of course, the path of a single molecule isn’t very significant — we need to repeat the simulation for many molecules, and average the results.

EXERCISE 4.28

Consider the diffusion of an aromatic molecule in air, having a velocity of 500 meters per second and a mean free path λ of 1 meter. Calculate the distance $\langle d \rangle$ a molecule moves in one second, averaging over 100 different molecules.

You probably found that the net displacement is much less than the 500 meters a free molecule would have traveled. We argued that this would be the case, as a consequence of the many collisions, but we have now successfully modeled that phenomenon on the computer. The importance of Monte Carlo

simulation is not the precision of the result, but the fact that it can yield qualitatively valid results in situations where any result is difficult to obtain.

Just how valid are the results? Certainly, the average displacement can be monitored as more molecular paths are considered, and some feel for the convergence of the results can be acquired, but perhaps a better approach is to monitor the *distribution* of displacements obtained. Physically, this distribution is akin to the density of aromatic molecules in the air at different distances from the perfume bottle, a distribution we clearly expect to be continuous. In our simulation, we have started all the molecules at the same instant, as if the perfume bottle were opened, many molecules allowed to escape, and then the bottle closed. After 1 second, and 500 collisions, we would expect very few molecules to still be in the vicinity of the origin. We would also expect few molecules to be found at large displacements since insufficient time has elapsed for them to travel very far. The distribution is thus expected to be small (or even zero) at the origin, to increase smoothly to a maximum and then to decrease smoothly to zero at large displacements. If the Monte Carlo sampling is sufficiently large, then the distribution we obtain should mimic this physically expected one. Conversely, if the the distribution is not realistic, then a larger sampling is needed.

EXERCISE 4.29

Investigate the distribution of net displacements as described. Plot a histogram indicating the number of paths yielding displacements between 0 and 1 m, between 1 and 2 m, and so on, using 100 different paths. Repeat the exercise, plotting histograms with 200, 300, 400, and 500 different paths, and compare the histograms obtained with what is expected on physical grounds.

Once we're satisfied that the method is working properly and that the results are statistically valid, we must ask ourselves if the results are *physically* valid. (Actually, this is a question we ask ourselves at every opportunity!) That the displacement is considerably less than the free displacement would have been is certainly expected, but what about the magnitude of the result itself? If you found, as I did, that after 1 second the net displacement is between 10 and 20 meters, then the the aroma of the perfume reaches the front of a 30-foot room in less than a second! That seems much too rapid. My own recollection of similar events is that it takes several minutes for the aroma to travel that far.

We're forced to conclude that the actual diffusion is slower than what we've found, which in turn means that the mean free path we've adopted is too large. We could repeat our simulation with a different value for the mean

free path, but there might be a better way. Consider: does the value of the mean free path really enter the calculation? Or have we actually evaluated something more universal than we thought: the net displacement, in units of the mean free path length, after 500 collisions? This is an example of a scaling relation, and is very important. If we can determine a fundamental relationship between the magnitude of the displacement, in units of the mean free path, and the number of collisions, then we can apply that relation to many different physical situations.

EXERCISE 4.30

Reexamine the simulation. Instead of monitoring the displacement after 1 second, monitor it after every collision, and average over a sufficiently large number of molecular paths to yield valid results. Plot the average net displacement as a function of the number of collisions.

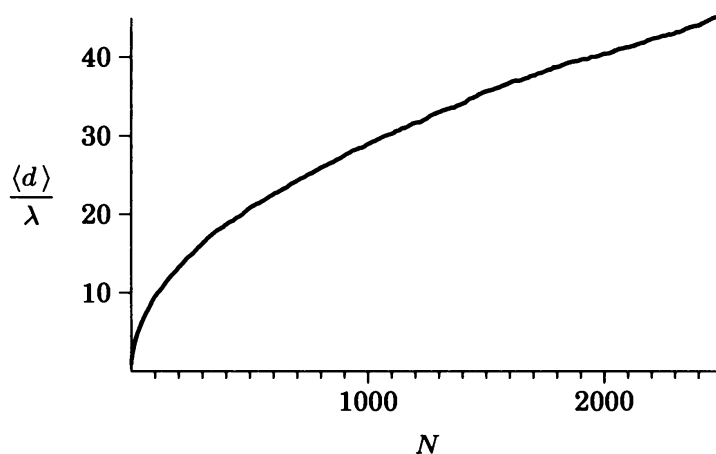


FIGURE 4.14 The average displacement $\langle d \rangle$ (in units of the mean free path λ) is plotted versus the number of steps taken. These data were obtained by averaging over 1000 different paths.

Averaging over 1000 molecular paths produced results presented in Figure 4.14. (This calculation takes several minutes but needs to be done only once.) The curve is remarkably smooth, and appears vaguely familiar, which suggests that further investigation might be worthwhile. Recall that if the displacement is proportional to a power of the number of collisions, N ,

$$\frac{\langle d \rangle}{\lambda} \approx N^q, \quad (4.135)$$

then

$$\ln \frac{\langle d \rangle}{\lambda} \approx q \ln N. \quad (4.136)$$

That is, a log-log plot of the data will be linear, and the slope of the line will be the power q . Such a plot is presented in Figure 4.15. Clearly, the plot is a linear one. With the data at $N = 10$ and $N = 1000$, the slope was determined ≈ 0.496 .

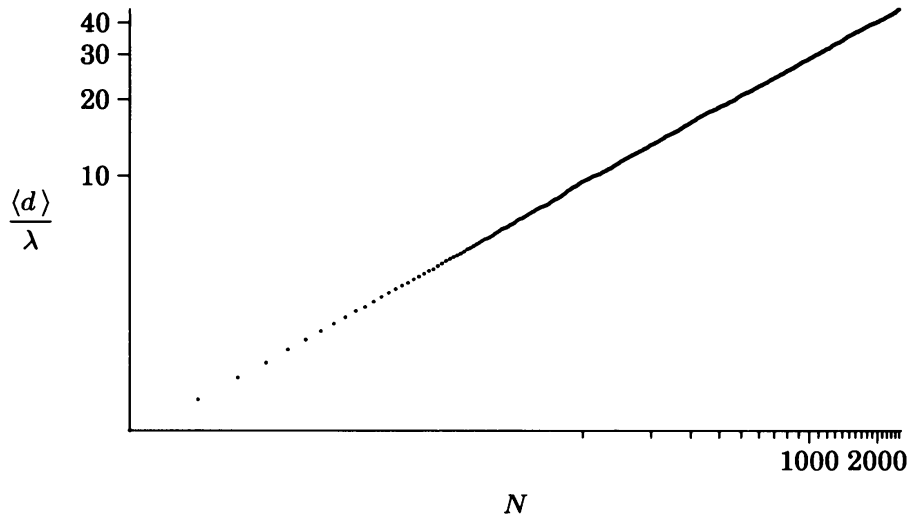


FIGURE 4.15 This is the same data plotted in Figure 4.14, but on a log-log scale. The apparent linearity of the curve suggests a simple power-law dependence.

There are certain numbers of which we should always be mindful: π , e , integers, and their reciprocals and powers. The slope we've determined is very close to $1/2$ — close enough to suspect that it's not accidental. We seem to have stumbled upon a fundamental truism, that the displacement is proportional to the square root of N ,

$$\frac{\langle d \rangle}{\lambda} \approx \sqrt{N}. \quad (4.137)$$

While we should always use analytic results to guide our computations, we should also be open to the possibility that our computations can lead us to new analytic results. Our results *do not prove* this relationship, of course, but strongly suggest that the relationship exists. At this juncture, we should set aside the computer and our computations, and research the availability of analytic derivations of this result. As Hamming, a pioneer in modern computing, has said, "The purpose of computing is insight, not numbers." It appears that we have gained significant insight from this simulation.

EXERCISE 4.31

Plot your data on a log-log scale, and verify the power dependence.

References

Since numerical integration is fundamental to many applications, accurate weights and abscissas for Gaussian integration were developed at nearly the same time as large-scale computers were becoming available to the scientific community. To avoid duplication of this effort, Stroud and Secrest published their results for all to use.

A. H. Stroud and Don Secrest, *Gaussian Quadrature Formulas*. Prentice-Hall, Englewood Cliffs, 1966.

An indispensable source of information regarding mathematical functions is the reference

Handbook of Mathematical Functions, edited by Milton Abramowitz and Irene A. Stegun, Dover Publications, New York, 1965.

Monte Carlo methods are becoming more widely used every day. A good introduction is provided by

Malvin H. Kalos and Paula A. Whitlock, *Monte Carlo Methods*, John Wiley & Sons, New York, 1986.

Some insight into the quality of random number generators can be discerned from the article

William H. Press and Saul A. Teukolsky, "Portable Random Number Generators," *Computers in Physics* **6**, 522 (1992).

Obtaining a Gaussian distribution of random numbers, rather than a uniform distribution, is discussed in

G. E. P. Box and M.E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.* **29**, 610 (1958).

Chapter 5:

Ordinary Differential Equations

To a large extent, the study of physics is the study of differential equations, so it's no surprise that the numerical solution of differential equations is a central issue in computational physics. The surprise is that so few traditional courses in physics, and even mathematics courses in differential equations, provide tools that are of practical use in solving real problems. The number of physically relevant "linear second-order homogeneous differential equations with constant coefficients" is not large; experience has led us to the conclusion that *all the interesting equations are either trivial, or impossibly difficult to solve analytically*. Traditional analysis solves the trivial cases, and can yield invaluable insight to the solution of the difficult ones. But the bottom line is that the difficult cases must be treated numerically.

You are probably familiar with the initial value problem, in which you have a differential equation such as $y''(x) = f(x, y', y'')$ and the initial conditions $y(0) = a$, $y'(0) = b$. Equations of this form are quite common, and we'll develop several methods suitable for their solution. There is another type of problem that is also of considerable importance to physics, which we'll also address in this chapter: the boundary value problem. Here, rather than having information about the derivative, you are told about the function at various points on the *boundary* of the integration region.

Unfortunately, there is no "best" method to solve all differential equations. Each equation has a character all its own, and a method that works well on one may work poorly, or even fail, on another. What we will do is to develop some general ideas concerning the numerical solution of differential equations, and implement these ideas in a code that will work reasonably well for a wide variety of problems. Keep in mind, however, that if you are faced with solving a difficult problem, say, one involving large sets of differential equations to be solved over a wide range of the independent variable, you

might be better off investigating a solution specifically tailored to the problem you're confronting.

Euler Methods

The most prolific mathematician of the eighteenth century, or perhaps of any century, was the Swiss-born Leonhard Euler (1707–1783). It has been said that Euler could calculate with no apparent effort, just as men breathe and eagles fly, and would compose his memoirs while playing with his 13 children. He lost the sight in his right eye when he was 28, and in the left when 59, but with no effect on his mathematical production. Aided by a phenomenal memory, and having practiced writing on a chalk board before becoming blind, he continued to publish his mathematical discoveries by dictating to his children. During his life, he published over 500 books and papers; the complete bibliography of Euler's work, including posthumous items, has 886 entries.

Euler made contributions to virtually every field of eighteenth-century mathematics, particularly the theory of numbers, and wrote textbooks on algebra and calculus. The prestige of his books established his notation as the standard; the modern usage of the symbols e , π , and i (for $\sqrt{-1}$) are directly attributable to Euler. He also made significant contributions in areas of applied mathematics: Euler wrote books on ship construction and artillery, on optics and music, and ventured into the areas of physics and astronomy. But our current interest, which represents only a minute fraction of Euler's output, is in methods due to him in solving differential equations.

Consider the differential equation

$$y'(x) = f(x, y). \quad (5.1)$$

(If f is a function of x alone, we can immediately “solve” for y :

$$y(x) = \int^x f(\chi) d\chi. \quad (5.2)$$

Since this is an “uninteresting” situation, we'll assume that f is a function of x and y .) Now, one might try to solve Equation (5.1) by Taylor's series; that is, if we knew all the derivatives, we could construct the solution from the expansion

$$y(x) = y(x_0) + (x - x_0)y'(x_0) + \frac{(x - x_0)^2}{2!}y''(x_0) + \cdots. \quad (5.3)$$

Since $y'(x)$ is known, we can obtain the higher derivatives — but it takes a little work. The second derivative, for example, is

$$\begin{aligned} y''(x) &= \frac{\partial}{\partial x} f(x, y) + \frac{dy}{dx} \frac{\partial}{\partial y} f(x, y) \\ &= \frac{\partial}{\partial x} f(x, y) + f(x, y) \frac{\partial}{\partial y} f(x, y) \end{aligned} \quad (5.4)$$

Clearly, this is leading to some complicated expressions, and the situation degenerates as we move to higher derivatives. As a practical matter, the Taylor series solution is not very helpful. However, it provides a standard against which other methods can be measured. To that end, we write

$$\begin{aligned} y(x) &= y_0 + (x - x_0)f(x_0, y_0) + \frac{(x - x_0)^2}{2!} \left[\frac{\partial f(x_0, y_0)}{\partial x} + f(x_0, y_0) \frac{\partial f(x_0, y_0)}{\partial y} \right] \\ &\quad + \frac{(x - x_0)^3}{3!} y'''(\xi), \end{aligned} \quad (5.5)$$

where $y_0 = y(x_0)$.

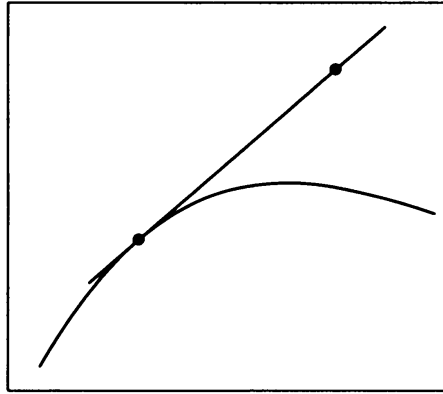


FIGURE 5.1 The *simple* Euler method.

The original differential equation gives us the derivative y' at any point; if we're given the value of y at some point x_0 , then we could approximate the function by a Taylor series, truncated to two terms:

$$y(x) \approx y(x_0) + (x - x_0)y'(x_0). \quad (5.6)$$

As simpleminded as it is, this method actually works! (But not well!) Denoting the size of the step $x - x_0$ by h , we can write Equation (5.6) as an equality

$$y(x_0 + h) = y(x_0) + hf(x_0, y(x_0)) = y_0 + hf_0, \quad (5.7)$$

where we've defined $y_0 = y(x_0)$ and $f_0 = f(x_0, y_0)$. This is known as the *simple Euler method*, and it allows us to move the solution along, one step at a time, as indicated in Figure 5.1. A typical implementation is to divide the total integration region into steps of size h , and to move the solution along one step at a time in the obvious way. As a check on accuracy, the calculation can be repeated for a different step size, and the results compared.

EXERCISE 5.1

Write a computer code to solve the differential equation

$$y'(x) = y^2 + 1$$

on the region $0 < x < 1$ using Euler's method, with $y(0) = 0$. Plot or graph your results for $h = 0.05, 0.10, 0.15$, and 0.20 , along with the exact result.

The problem with the simple Euler method is that the derivative at the beginning of the interval is assumed constant over the entire step; the derivative at the end of the interval is not used (in this interval). But we've already seen that such asymmetric treatments always lead to low levels of accuracy in the solution. Wouldn't it be better to use some *median* value of the derivative, say, the value halfway through the step? Of course — but how is the derivative at the midpoint evaluated, when the derivative is itself a function of y ? Good question.

Let's use Euler's method to give us a *guess* at what the solution should be at the midpoint, $x_{mid} = x_0 + \frac{h}{2}$. That is,

$$y(x_{mid}) = y_0 + \frac{h}{2}y'_0 = y_0 + \frac{h}{2}f_0, \quad (5.8)$$

where we've again associated the derivative of y with the function f — that is, we've used the differential equation we're trying to solve. With this expression for $y(x_{mid})$, we can evaluate the derivative at the midpoint, $f(x_{mid}, y_{mid})$, and using *that* as our approximation to the derivative over the entire interval we find

$$y(x_0 + h) = y(x_0) + hf(x_{mid}, y_{mid}) \quad (5.9)$$

This is the *modified Euler's method*, and has an interesting geometrical interpretation. (See Figure 5.2.) While Euler's method corresponds to drawing a straight line with derivative $f(x_0, y_0)$ through the point (x_0, y_0) , the modified Euler's method puts a line through (x_0, y_0) , but with (approximately) the derivative at the midpoint of the interval. Another way of thinking of this

method is to consider a simple approximation to the derivative at the midpoint,

$$f(x_{mid}, y_{mid}) = y'(x_{mid}) \approx \frac{y(x_0 + h) - y(x_0)}{h}. \quad (5.10)$$

Using Euler's method to approximate y_{mid} , the modified Euler method quickly follows.

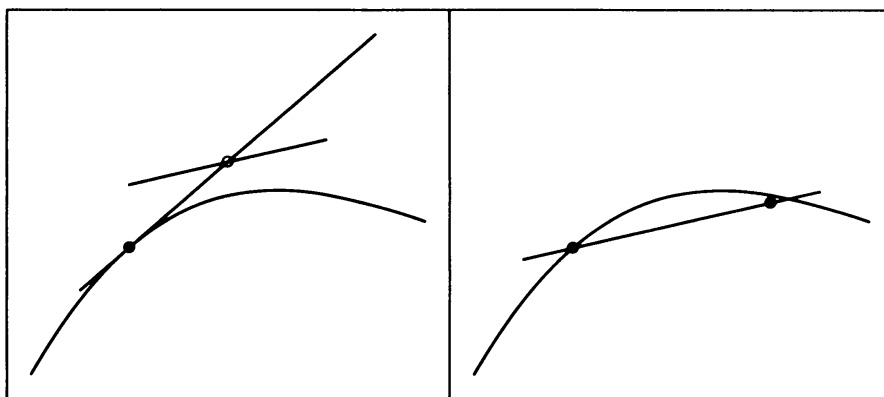


FIGURE 5.2 The *modified* Euler method.

Yet another variation of Euler's method is possible if we attempt a solution using a *mean* value of the derivative. (See Figure 5.3.) That is, we use Euler's equation to guess at $y(x_0 + h)$, which we use to evaluate the derivative at the end of the interval. This derivative is averaged with the "known" derivative at the start of the interval, and this mean derivative is used to advance the solution. The *improved Euler method* is thus given as

$$y(x_0 + h) = y(x_0) + h \frac{f_0 + f(x_0 + h, y_0 + hf_0)}{2}. \quad (5.11)$$

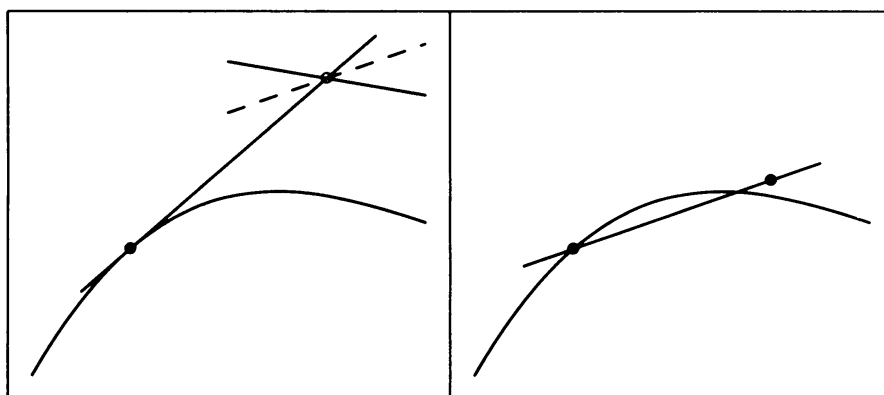


FIGURE 5.3 The *improved* Euler method.

EXERCISE 5.2

Modify your computer code to solve differential equations by the modified Euler and improved Euler methods, and solve the equation

$$y'(x) = y^2 + 1, \quad y(0) = 0$$

using a step size $h = 0.10$. Prepare a table of solutions comparing the three methods, on the interval $0 < x < 1$.

Constants of the Motion

In Exercise 5.1, you probably found that the solution improves as the step size is made smaller, but that the approximations always lag behind the exact solution. Using the modified or improved methods gives better results, based on a comparison among approximations or against the exact result. But before we explore even better approximations, we should note that there are other ways to judge the quality of a solution: on physical grounds, it might happen that a particular quantity is conserved. In that situation, the degree to which that quantity is calculated to be constant is indicative of the quality of the solution. For example, consider a mass on a spring — the velocity of the mass is determined from the equation

$$\frac{dv}{dt} = a = \frac{F}{m} = \frac{-kx}{m}. \quad (5.12)$$

For simplicity, take the mass and the force constant to equal 1, so that we have

$$\frac{dv}{dt} = -x. \quad (5.13)$$

From the definition of velocity, we also have

$$\frac{dx}{dt} = v. \quad (5.14)$$

For a time step δ , the simple Euler approximation to the solution to this set of equations is simply

$$v(t_0 + \delta) = v(t_0) + \delta \left. \frac{dv}{dt} \right|_{t_0} = v(t_0) - \delta x(t_0), \quad (5.15)$$

$$x(t_0 + \delta) = x(t_0) + \delta \left. \frac{dx}{dt} \right|_{t_0} = x(t_0) + \delta v(t_0). \quad (5.16)$$

But the spring — at least, an ideal one — provides a conservative force so that the energy of the system, $E = mv^2/2 + kx^2/2$, is a constant of the motion. Imagine that the system is set in motion at $t = 0$, $x = 0$ with $v = 1$. Using time steps of $\delta = 0.1$, the equations can be solved, and the “solutions” thus determined. At each step, we can derive the energy from the calculated positions and velocities, and exhibit these quantities as in Figure 5.4. Needless to say, something is not working here.

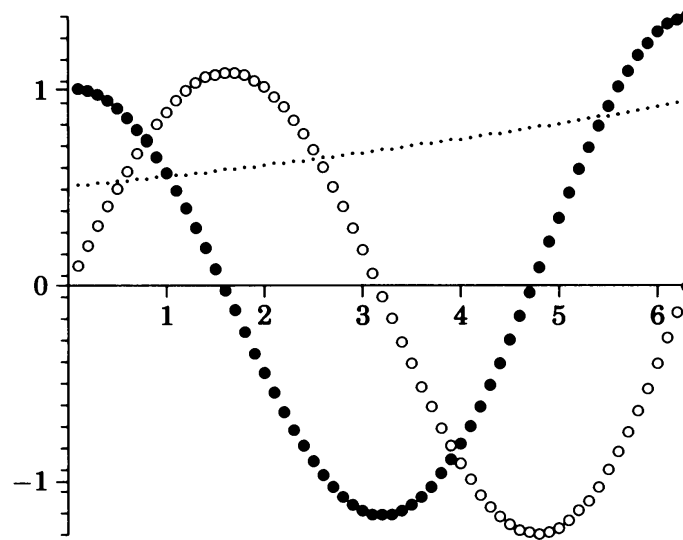


FIGURE 5.4 Position (\circ), velocity(\bullet), and energy(\cdot) as a function of time, for the simple harmonic oscillator, as calculated by the simple Euler method.

But we already knew that the simple Euler method had its problems; do the modified or improved methods do a better job? Applying the modified Euler method to Equations (5.13) and (5.14), we first use the simple Euler’s method to approximate v and x halfway through the time increment,

$$v(t_0 + \frac{\delta}{2}) = v(t_0) + \frac{\delta}{2} \left. \frac{dv}{dt} \right|_{t_0} = v(t_0) - \frac{\delta}{2} x(t_0), \quad (5.17)$$

$$x(t_0 + \frac{\delta}{2}) = x(t_0) + \frac{\delta}{2} \left. \frac{dx}{dt} \right|_{t_0} = x(t_0) + \frac{\delta}{2} v(t_0). \quad (5.18)$$

These values are then used over the entire time increment to determine v and

x at $t = t_0 + \delta$:

$$v(t_0 + \delta) = v(t_0) + \delta \left. \frac{dv}{dt} \right|_{t_0 + \frac{\delta}{2}} = v(t_0) - \delta x(t_0 + \frac{\delta}{2}), \quad (5.19)$$

$$x(t_0 + \delta) = x(t_0) + \delta \left. \frac{dx}{dt} \right|_{t_0 + \frac{\delta}{2}} = x(t_0) + \delta v(t_0 + \frac{\delta}{2}). \quad (5.20)$$

The code to implement the modified Euler method is only a slight modification of that used for the simple Euler method — simply add the evaluation of `xmid` and `vmid` to the code, and use these values to evaluate `xnew` and `vnew` from the previous `xold` and `vold` values.

EXERCISE 5.3

Use the modified Euler method with a time step of 0.1 to solve the “mass on a spring” problem, and present your results in a plot similar to Figure 5.4. How well is the energy conserved?

It might seem that a numerical method that preserves constants of the motion is inherently “better” than one that does not. Certainly, the improved and modified Euler methods are to be preferred over the simple Euler method. But this preference is derived from improvements made in the algorithm, as *verified* by the computation of constants of the motion, not because the constants were *guaranteed* to be preserved.

It is possible to *construct* an algorithm that preserves constants of the motion. For example, consider the mass-on-a-spring problem. We might use the simple Euler expression to determine position,

$$x(t_0 + \delta) = x(t_0) + \delta \left. \frac{dx}{dt} \right|_{t_0} = x(t_0) + \delta v(t_0), \quad (5.21)$$

and determine the velocity by *requiring* that

$$E = \frac{mv^2}{2} + \frac{kx^2}{2}. \quad (5.22)$$

This will give us the *magnitude* of the velocity, and we could obtain its *sign* by requiring it to be the same as that obtained from the expression

$$v(t_0 + \delta) = v(t_0) + \delta \left. \frac{dv}{dt} \right|_{t_0} = v(t_0) - \delta x(t_0). \quad (5.23)$$

This algorithm is *absolutely guaranteed* to conserve energy, within the computer's ability to add and subtract numbers. But how good is it otherwise?

EXERCISE 5.4

Use this “guaranteed energy-conserving” algorithm to solve the mass-on-a-spring problem, and plot the results. How does it do?

Runge–Kutta Methods

The Euler methods are examples of a general class of approximations known as Runge–Kutta methods, characterized by expressing the solution in terms of the derivative $f(x, y)$ evaluated with different arguments. This is in contrast to the Taylor's series solution which requires many different derivatives, all evaluated with the same arguments. Runge–Kutta methods are extremely popular, in part due to the ease with which they can be implemented on computers.

We notice that all the Euler methods can be written in the form

$$y(x_0 + h) = y(x_0) + h[\alpha f(x_0, y_0) + \beta f(x_0 + \gamma h, y_0 + \delta h f_0)], \quad (5.24)$$

Let's see how well this expression agrees with Taylor's series. A function $f(x, y)$ of two variables can be expanded as

$$\begin{aligned} f(x, y) = & f(x_0, y_0) + (x - x_0) \frac{\partial f(x_0, y_0)}{\partial x} + (y - y_0) \frac{\partial f(x_0, y_0)}{\partial y} \\ & + \frac{(x - x_0)^2}{2} \frac{\partial^2 f(\xi, \eta)}{\partial x^2} + (x - x_0)(y - y_0) \frac{\partial^2 f(\xi, \eta)}{\partial x \partial y} \\ & + \frac{(y - y_0)^2}{2} \frac{\partial^2 f(\xi, \eta)}{\partial y^2} + \dots, \end{aligned} \quad (5.25)$$

where $x_0 \leq \xi \leq x$ and $y_0 \leq \eta \leq y$. Using this expression to expand $f(x_0 + \gamma h, y_0 + \delta h f_0)$ of Equation (5.24), we find that

$$\begin{aligned} y(x) = & y_0 + h\alpha f(x_0, y_0) \\ & + h\beta \left[f(x_0, y_0) + h\gamma \frac{\partial f(x_0, y_0)}{\partial x} + h\delta f(x_0, y_0) \frac{\partial f(x_0, y_0)}{\partial y} + O(h^2) \right] \end{aligned}$$

$$\begin{aligned}
 &= y_0 + h(\alpha + \beta)f(x_0, y_0) \\
 &\quad + h^2\beta \left[\gamma \frac{\partial f(x_0, y_0)}{\partial x} + \delta f(x_0, y_0) \frac{\partial f(x_0, y_0)}{\partial y} \right] + O(h^3)
 \end{aligned} \tag{5.26}$$

This expression agrees with the Taylor series expression of Equation (5.5) through terms involving h^2 , if we require that

$$\begin{aligned}
 \alpha + \beta &= 1, \\
 \beta\gamma &= 1/2, \\
 \text{and } \beta\delta &= 1/2
 \end{aligned} \tag{5.27}$$

Thus the improved and modified Euler methods both agree with the Taylor series through terms involving h^2 , and are said to be second-order Runge–Kutta methods. Although these equations require that $\gamma = \delta$, otherwise there is considerable flexibility in choosing the parameters; the optimum second-order method, in the sense that the coefficient multiplying the h^3 term is minimized, has $\alpha = 1/3$, $\beta = 2/3$, and $\gamma = \delta = 3/4$.

While the Euler method jumps right in to find a solution, the improved and modified methods are more conservative, testing the water (so to speak) before taking the plunge. These methods can actually be derived in terms of an integral: since $y' = f(x, y)$, then clearly

$$y(x_0 + h) = y(x_0) + \int_{x_0}^{x_0+h} f(\tau, y) d\tau. \tag{5.28}$$

The only problem, of course, is that the sought-after solution y appears under the integral on the right side of the equation, as well as on the left side of the equation. Approximating the integral by the midpoint rule, we have

$$y(x_0 + h) = y(x_0) + hf(x_0 + \frac{h}{2}, y_{mid}). \tag{5.29}$$

y_{mid} is then approximated by a Taylor series expansion,

$$y_{mid} \approx y(x_0) + \frac{h}{2}f(x_0, y_0). \tag{5.30}$$

Since the integral is already in error $O(h^2)$, there is no point in using a more accurate series expansion. With these approximations, Equation (5.28) then reads

$$y(x_0 + h) = y(x_0) + hf(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f_0), \tag{5.31}$$

which we recognize as the modified Euler method. In a similar fashion, the improved Euler method can be derived by approximating the integral in Equation (5.28) by the trapezoid rule.

The methods we've outlined can be used to derive higher order Runge–Kutta methods. Perhaps the most popular integration method ever devised, the fourth-order Runge–Kutta method, is written in terms of intermediate quantities defined as

$$\begin{aligned} f_0 &= f(x_0, y_0), \\ f_1 &= f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f_0\right), \\ f_2 &= f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f_1\right), \\ f_3 &= f(x_0 + h, y_0 + hf_2). \end{aligned} \tag{5.32}$$

The solution is then expressed as

$$y(x_0 + h) = y(x_0) + \frac{h}{6}(f_0 + 2f_1 + 2f_2 + f_3). \tag{5.33}$$

This is the standard, classic result often referred to as simply *the* Runge–Kutta method, and is a mainstay in the arsenal of numerical analysts. For the special case that $f = f(x)$, this result is obtained by evaluating the integral of Equation (5.28) by Simpson's rule.

EXERCISE 5.5

One of the standard problems of first-year physics is one-dimensional projectile motion — but contrary to standard practice, let's include air resistance to see how large an effect it is. The time rate of change of the momentum is

$$\frac{dp}{dt} = mg - kv^2,$$

where m is the mass of the object, $g = 9.8 \text{ m/s}^2$ is the acceleration due to gravity, and k is a drag coefficient. For a particular sphere of mass 10^{-2} kg the drag coefficient was determined to be $k = 10^{-4} \text{ kg/m}$. Letting $p = mv$, use the fourth-order Runge–Kutta method to find the velocity of the sphere released from rest as a function of time for $0 < t < 10$ seconds. Choose a step size to ensure 4-significant-digit accuracy. Compare your calculation to the zero-th order approximation, e.g., the analytic solution obtained by ignoring air resistance.

Adaptive Step Sizes

In the previous exercise, a typical problem involving ordinary differential equations was presented. Part of the specification of that exercise was the stipulation that a given level of accuracy in the solution be achieved. How did you achieve that accuracy, and — before reading further — are you convinced that your solution is *really* that accurate?

Probably the most common way to ascertain the accuracy of a solution is to calculate it twice, with two different step sizes, and compare the results. This comparison could be made after many propagation steps, i.e., after integrating the solution for some distance. But the nature of the solution might change considerably from one region to another — a smaller step size might be necessary *here* and not *there* — so that the calculated results should be compared often, allowing for a decrease (or increase!) of the step size where appropriate.

By far the easiest way to accomplish this comparison is to use step sizes h and $h/2$, and compare immediately — if the difference is small, then the error is assumed small. In fact, this estimate of the error is used to adjust the size of the step. If the error is larger than tolerated, then the step size is halved. Likewise, if the error is less than some predetermined value, the steps are too small and too much work is being performed; the step size is increased. Such an adaptive step size modification to the classic Runge–Kutta method greatly enhances its utility, so much so that methods without some form of adaptation simply should not be used.

An additional benefit of having two approximations to the result is that Richardson’s extrapolation can be used to obtain a “better” estimate of the solution. Since the Runge–Kutta method is accurate through h^4 , the two solutions can be combined to eliminate the first term of the error ($\sim h^5$): if $y_{(h)}$ and $y_{(h/2)}$ are the two solutions, the extrapolated value is

$$y_{\text{extrapolated}} = \frac{16y_{(h/2)} - y_{(h)}}{15}. \quad (5.34)$$

EXERCISE 5.6

Modify your Runge–Kutta program to take adaptive step sizes, and to improve upon the results at each step via Richardson’s extrapolation. Use this modified code to solve the projectile problem of the previous exercise.

Runge–Kutta–Fehlberg

Rather than interval halving/doubling, there's another, even more interesting, way that we can utilize our knowledge of the error to our benefit. Let's see if we can devise a scheme by which we can maintain a given accuracy for the derivative at each step in the solution of the differential equation. We'll denote the tolerated error in the derivative by ε , so that the acceptable error in the function is $h\varepsilon$. For an n -th order Runge–Kutta solution we have

$$y(x_0 + h) = y_{exact} + kh^{n+1}, \quad (5.35)$$

where y_{exact} is the exact solution, while for an $(n + 1)$ -th order solution we have

$$\hat{y}(x_0 + h) = y_{exact} + \hat{k}h^{n+2}. \quad (5.36)$$

The difference between these two solutions is simply

$$y(x_0 + h) - \hat{y}(x_0 + h) = kh^{n+1} - \hat{k}h^{n+2} \approx kh^{n+1}, \quad (5.37)$$

where the approximation is valid for small h . We can then solve for k ,

$$k \approx \frac{y - \hat{y}}{h^{n+1}}. \quad (5.38)$$

But the difference between the two solutions is also a measure of the error, which is to be maintained at the level $h\varepsilon$. Let h_{new} be a new step size, for which these two expressions for the error agree. That is, we'll *require* that

$$h_{new}\varepsilon = kh_{new}^{n+1} = \frac{h_{new}^{n+1}}{h^{n+1}}|y - \hat{y}|. \quad (5.39)$$

Equation (5.39) is easily solved for h_{new} , with the result

$$h_{new} = h \sqrt[n]{\frac{h\varepsilon}{|y(x_0 + h) - \hat{y}(x_0 + h)|}}. \quad (5.40)$$

Now, we need to interpret this result just a bit. With the step size h , both y and \hat{y} can be calculated, and so a direct numerical estimate of the error $|y - \hat{y}|$ can be calculated. But we know how this error depends on h , and so can calculate the coefficient k from Equation (5.38). And knowing k allows us to evaluate an h_{new} that would have given an error in the derivative of only ε . *Voila!* If the calculated error is greater than the acceptable error, then too large a step will have been taken and h will be greater than h_{new} . Since the error has been determined to be larger than acceptable, we'll repeat

the step using a smaller step size. On the other hand, it can happen that the calculated error is smaller than we've specified as acceptable, h is less than h_{new} , so that we could have taken a larger step. Of course, it would be wasteful actually to repeat the step — but we can use h_{new} as a guess for the next step to be taken! This leads to a very efficient algorithm in which the step size is continually adjusted so that the actual error is near, but always less than, the prescribed tolerance. Beginning with an initial value of $y(x_0)$ and an initial h , the algorithm consists of the following steps:

1. Calculate $y(x_0 + h)$ and $\hat{y}(x_0 + h)$ from $y(x_0)$.
2. Calculate h_{new} . If h_{new} is less than h , *reject* the propagation to $x_0 + h$, redefine h , and repeat step 1. If h_{new} is greater than h , *accept* the propagation step, replace x_0 by $x_0 + h$, redefine h , and go to step 1 to continue propagating the solution.

Since the cost of repeating a step is relatively high, we'll intentionally be conservative and use a step size somewhat smaller than that predicted, say, only 90% of the value, so that corrective action is only rarely required.

So far so good, but it's not obvious that much effort, if any, has been saved. Fehlberg provided the real key to this method by developing Runge-Kutta methods of different order that use *exactly the same* intermediate function evaluations. Once the first approximation to the solution has been found, it's trivial to calculate the second. The method we will use is the fourth-order/fifth-order method, defined in terms of the following intermediate function evaluations:

$$f_0 = f(x_0, y_0), \quad (5.41)$$

$$f_1 = f(x_0 + \frac{h}{4}, y_0 + \frac{h}{4}f_0), \quad (5.42)$$

$$f_2 = f(x_0 + \frac{3h}{8}, y_0 + \frac{3h}{32}f_0 + \frac{9h}{32}f_1), \quad (5.43)$$

$$f_3 = f(x_0 + \frac{12h}{13}, y_0 + \frac{1932h}{2197}f_0 - \frac{7200h}{2197}f_1 + \frac{7296h}{2197}f_2), \quad (5.44)$$

$$f_4 = f(x_0 + h, y_0 + \frac{439h}{216}f_0 - 8hf_1 + \frac{3680h}{513}f_2 - \frac{845h}{4104}f_3), \quad (5.45)$$

$$f_5 = f(x_0 + \frac{h}{2}, y_0 - \frac{8h}{27}f_0 + 2hf_1 - \frac{3544h}{2565}f_2 + \frac{1859h}{4104}f_3 - \frac{11h}{40}f_4). \quad (5.46)$$

With these definitions, the fourth-order approximation is given as

$$y = y_0 + h(\frac{25}{216}f_0 + \frac{1408}{2565}f_2 + \frac{2197}{4104}f_3 - \frac{1}{5}f_4) \quad (5.47)$$

and the fifth-order one as

$$\hat{y} = y_0 + h\left(\frac{16}{135}f_0 + \frac{6656}{12825}f_2 + \frac{28561}{56430}f_3 - \frac{9}{50}f_4 + \frac{2}{55}f_5\right). \quad (5.48)$$

The error can be evaluated directly from these expressions,

$$Err = \hat{y} - y = h\left(\frac{1}{360}f_0 - \frac{128}{4275}f_2 - \frac{2197}{75240}f_3 + \frac{1}{50}f_4 + \frac{2}{55}f_5\right), \quad (5.49)$$

so that y need never be explicitly calculated. Since we're using a fourth-order method, an appropriate (conservative) expression for the step size is

$$h_{new} = 0.9h \sqrt[4]{\frac{|h|\varepsilon}{|y(x_0 + h) - \hat{y}(x_0 + h)|}}. \quad (5.50)$$

The computer coding of these expressions is actually quite straightforward, although a little tedious. In particular, the coefficients in the expressions must be entered very carefully. In the following computer code, these coefficients are specified separately from their actual use, and most are specified by expressions involving operations as well as numerical values. Twelve divided by thirteen is not easily expressed as a decimal, and should be expressed to the full precision of the computer if it were; it's easier, and more accurate, to let the computer do the division. By using symbolic names in the actual expressions of the algorithm, the clarity of the code is enhanced. And by placing the evaluation of the coefficients within a `PARAMETER` statement, we are assured that they won't be "accidentally" changed. Some of the computer code might look like the following:

```

Subroutine R_K_F
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This program solves differential equations by the
* Runge-Kutta-Fehlberg adaptive step algorithm.
*                                     11/8/86
*
      double precision h, x, y, x0, y0, ...
*
* Specify error tolerance.
*
      double precision Epsilon
      Parameter ( Epsilon=1.d-5 )

```

```

*
* Specify coefficients used in the R-K-F algorithm:
*
* The coefficients Am are used to determine the 'x' at
* which the derivative is evaluated.
*
      double precision a1,a2,a3,a4,a5
      parameter (a1=0.25D0, a2=0.375D0, a3=1.2D1/1.3D1,...
*
* The coefficients Bmn are used to determine the 'y' at
* which the derivative is evaluated.
*
      double precision b10, b20,b21, b30,b31,b32,...
      parameter (b10=0.25D0, b20=3.D0/3.2D1,...
*
* The Cn are used to evaluate the solution YHAT.
*
      double precision c0,c2,c3,c4,c5
      parameter( c0=1.6D1/1.35D2, c2=6.656D3/1.2825D4,...
*
* The Dn are used to evaluate the error.
*
      double precision d0,d2,d3,d4,d5
      parameter( d0 = 1.d0/3.6D2, d2 = -1.28D2/4.275D3,...
*
* Initialize the integration:
*
      h = ...
      x0 = ...
      y0 = ...
*
* The current point is specified as (X0,Y0) and the
* step size to be used is H. The function DER evaluates
* the derivative function at (X,Y). The solution is
* moved forward one step by:
*
100      f0= der(x0,y0)

200      x = x0 + a1*h
          y = y0 + b10*h*f0
          f1 = der(x,y)

          x = x0 + a2*h

```

```

y = y0 + b20*h*f0 + b21*h*f1
f2 = der(x,y)

x = x0 + a3*h
y = y0 + h*(b30*f0 + b31*f1 + b32*f2)
f3 = der(x,y)

x = x0 + a4*h
y = y0 + h*(b40*f0 + b41*f1 + b42*f2 + b43*f3)
f4 = der(x,y)

x = x0 + a5*h
y = y0 + h*(b50*f0 + b51*f1 + b52*f2 + b53*f3
+          + b54*f4)
f5 = der(x,y)

yhat = y0 + h * ( c0*f0 + c2*f2 + c3*f3
+                + c4*f4 + c5*f5 )

err = H * DABS ( d0*f0 + d2*f2 + d3*f3
+               + d4*f4 + d5*f5 )

MaxErr = h*epsilon
hnew = 0.9d0 * h * sqrt( sqrt(MaxErr/Err) )

*
* If the error is too large, repeat the propagation step
* using HNEW.
*
      IF( Err .gt. MaxErr ) THEN
        h = hnew
        goto 200
      ENDIF

*
* The error is small enough, so this step is acceptable.
* Redefine X0 to move the solution along; let the more
* accurate approximation YHAT become the initial value
* for the next step.
*
      x0 = x0 + h
      y0 = yhat
      h = hnew

*
* Have we gone far enough? Do we stop here? What gives?
*
```

```

        if 'we take another step' goto 100
        ...
    end
* -----
    double precision function DER(x,y)
    double precision x,y
    der = ...
    end

```

For the moment, you should just loop through this code until x is greater than the maximum desired, at which time the program should terminate. As it stands, there is no printout, which you'll probably want to change.

■ EXERCISE 5.7

Repeat the problem of projectile motion with air resistance, with all the parameters the same, but using the Fehlberg algorithm. You can start the integration with almost any h — if it's too large, the program will redefine it and try again, and if it's too small, well, there's no harm done and the second step will be larger. Use a maximum error tolerance of 10^{-5} , and print out the accepted x and y values at each step.

Before we can go much farther in our investigation of the solutions of differential equations, we need to make some modifications to our Fehlberg integrator. By this time, you've probably found that the integrator works very well, and you might suspect that further refinement isn't necessary. And you're probably right! But we want to develop a code that is going to do its job, without a lot of intervention from us, for a large variety of problems. That is, we want it to be smart enough to deal with most situations, and even more important, smart enough to tell us when it can't!

One of the things that can happen, occasionally, is that the *very nature* of the solution can change; in such a case, it's possible for the integrator to “get confused.” To avoid this problem, we will impose an additional constraint upon the step size predictions: the step size shouldn't change too much from one step to the next. That is, if the prediction is to take a step 100 times greater than the current one, then it's a good bet that something “funny” is going on. So, we won't allow the prediction to be larger than a factor of 4 times the current size. Likewise, we shouldn't let h decrease too rapidly either; we'll only accept a factor of 10 decrease in h . (While it might appear that overriding the step size prediction would automatically introduce an error, such is not the case since the accuracy of the step is verified after its completion and repeated if the error isn't within the specified limit. It does

insure that the integrator isn't tricked into taking *too small* of a step.) And while we're at this, let's impose maximum and minimum step sizes: if the predicted h exceeds h_{max} , h will simply be redefined; but if h falls less than h_{min} , it's probably an indication that something is seriously amiss, and the integration should be stopped and an appropriate message displayed. These modifications should be made to your code, so that the result looks something like the following:

```

...
Hmax = ...
H = Hmax
Hmin = ...
...
Hnew= 0.9d0 * h * sqrt( sqrt(MaxErr/Err) )
*
* Check for increase/decrease in H, as well as H-limits.
*
IF( Hnew .gt. 4.0d0*H)Hnew = 4.d0*H
IF( Hnew .lt. 0.1d0*H)Hnew = .1d0*H
IF( Hnew .gt. Hmax) Hnew = Hmax
IF( Hnew .lt. Hmin) THEN
    write(*,*) ' H is TOO SMALL'
    write(*,*)X0,Y0,Yhat,H,Hnew
    STOP 'Possible Problem in RKF Integrator'
ENDIF
*
* If the error is too large, REPEAT the current
* propagation step with HNEW.
*
IF( Err .gt. MaxErr ) THEN
    ...

```

Note that we can conveniently initialize the integrator with $h = h_{max}$. These modifications to $Hnew$ *do not* change the conditions under which the “current” step is accepted or rejected; they simply restrict the range of acceptable step sizes. After making these modifications, check them on the following project.

EXERCISE 5.8

Attempt to find the solution of

$$y' = \frac{1}{x^2}, \quad y(-1) = 1$$

on the interval $-1 \leq x \leq 1$. Take $h_{max} = 0.1$, $h_{min} = 0.001$, and $\text{Epsilon} = 5 \times 10^{-5}$.

Second-Order Differential Equations

So far we've been quite successful in finding numerical solutions to our differential equations. However, we've mostly been concerned with first-order equations of the form

$$y' = f(x, y). \quad (5.51)$$

In physics, *second-order* equations are much more important to us in that they describe a much larger number of real physical systems. We thus need to work with equations of the form

$$y'' = f(x, y, y'). \quad (5.52)$$

Now, it might appear that this is a totally new and different problem. There are, in fact, numerical approaches specifically developed to treat second-order equations. However, it is also possible to *reduce* this problem to a more familiar form.

In fact, we've already done this when we looked at the “mass-on-the-spring” problem. You'll recall that we were able to treat the problem by considering two quantities, the position and the velocity, each of which was governed by a first order differential equation but involving both quantities. Although our notation was clumsy, we were able to solve the problem.

To facilitate the general solution to Equation (5.52), let's introduce some new variables; in particular, let's let $y_1 = y$, and $y_2 = y'$. Aside from the computational advantages which we're seeking, this just makes a lot of sense: y' is a *different thing* than is y — just as position and velocity are different — and so why shouldn't it have it's own name and be treated on an equal basis with y ? We then find that the original second-order differential equation can be written as a set of two, first-order equations,

$$y'_1 = y_2, \quad (5.53)$$

$$y'_2 = f(x, y_1, y_2). \quad (5.54)$$

At this point, we could go back and rederive all our methods — Euler, Runge-Kutta, and Fehlberg — but we don't have to do that. Instead, we note that these look like equations involving *vector* components, an observation further enhanced if we define

$$f_1 = y_2, \quad (5.55)$$

and

$$f_2 = f(x, y_1, y_2). \quad (5.56)$$

Our equations can then be written in vector form

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ f(x, y_1, y_2) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \quad (5.57)$$

or

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \quad (5.58)$$

or

$$\mathbf{y}' = \mathbf{f},$$

where \mathbf{y} and \mathbf{f} denote vectors. We thus see that this problem *is not* fundamentally different from the first-order problem we've been working with — it just has more components! Instead of stepping our one solution along, we need to modify the existing code so that it steps our two components along. We thus see that the modifications in our existing code are rather trivial, and consist primarily of replacing scalar quantities by arrays. The independent variable X is still a scalar, but the dependent variables Y and Y_{hat} , and the intermediate quantities F_0, F_1, \dots , all become dimensioned arrays. A sample of the changes to be made are illustrated below:

```
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This program solves SECOND-ORDER differential equations
* by the Runge-Kutta-Fehlberg adaptive step algorithm.
* Note that the solution and the intermediate function
* evaluations are now stored in ARRAYS, and are treated
* as components of VECTORS.
*
*                                     11/9/87
*
*
*       double precision h, x, y(2), x0, y0(2), ...
*       ...
*
* Specify coefficients used in the R-K-F algorithm:
*
* The coefficients Am are used to determine the 'x' at
* which the derivative is evaluated.
*
*       double precision a1,a2,a3,a4,a5
*       parameter (a1=0.25D0, a2=0.375D0, a3=1.2D1/1.3D1,...
```

```

...
*
* Initialize the integration:
*
    h = ...
    x0 = ...
    y0(1) = ...
    y0(2) = ...
*
*
*****
* Remember: Y, Y0, YHAT, and all the intermediate *
* quantities F0, F1, etc., are dimensioned arrays! *
*****
*
* The current point is specified as (x0,Y0) and the
* step size to be used is H. The subroutine DERS evaluates
* all the derivatives, e.g., all the components of the
* derivative vector, at (X,Y). The solution is moved
* forward one step by:
*
100    call DERS( x0, y0, f0)

200    x = x0 + a1*h
      DO i = 1, 2
        y(i) = y0(i) + b10*h*f0(i)
      END DO
      call DERS( x, y, f1)

      x = x0 + a2*h
      DO i = 1, 2
        y(i) = y0(i) + b20*h*f0(i) + b21*h*f1(i)
      END DO
      call DERS( x, y, f2)
      ...

      BigErr = 0.d0
      DO i = 1, 2
        Yhat(i) = Y0(i)+H*( C0*F0(i) + C2*F2(i) + C3*F3(i)
+                          + C4*F4(i) + C5*F5(i) )
        Err      = H * DABS( D0*F0(i) + D2*F2(i) + D3*F3(i)
+                          + D4*F4(i) + D5*F5(i) )
      IF(Err .gt. BigErr)BigErr = Err
      END DO

```

```

      MaxErr = h*epsilon
      hnew = 0.9d0 * h * sqrt( sqrt(MaxErr/BigErr) )
*
* Check for increase/decrease in H, as well as H-limits.
*
      if( Hnew .gt. 4.d0*H)Hnew = ...
      ...
*
* If the error is too large, repeat the propagation step
* using HNEW.
*
      IF( BigErr .gt. MaxErr ) THEN
        h = hnew
        goto 200
      ENDIF
*
* The error is small enough, so this step is acceptable.
* Redefine X0 to move the solution along; let the more
* accurate approximation become the initial value for
* the next step.
*
      x0 = x0 + h
      DO i = 1, 2
        y0(i) = yhat(i)
      END DO
      h = hnew
*
* Have we gone far enough? Do we stop here? What gives?
*
      if 'we should take another step' goto 100
      ...

    end
*-----
    Subroutine DERS (x, y, f)
*
* This subroutine evaluates all the components of the
* derivative vector, putting the results in the array 'f'.
*
      double precision x, y(2), f(2)
      f(1) = y(2)
      f(2) = < the second derivative function
               of Equation (5.56) >
    end

```

The biggest change, of course, is in the dimensioning of the relevant quantities. Note that we must also change the error criteria somewhat — we are interested in the *largest error* in any of the components, and so introduce the variable `BigErr` to keep track of it. We've also put almost all the information about the particular differential equation to be solved into the subroutine `DERS`, which calculates all the components of the derivative vector. At this juncture, it would not be a major modification to write the Runge–Kutta–Fehlberg algorithm as an independent subroutine, not tied to a specific problem being solved.

■ EXERCISE 5.9

Make the necessary modifications to your code, and test it on the differential equation

$$y'' = -4y, \quad y(0) = 1, \quad y'(0) = 0,$$

on the interval $0 \leq x \leq 2\pi$. Compare to the analytic result. Use the computer to plot these results as well — the visualization of the result is much more useful than a pile of numbers in understanding what's going on.

The Van der Pol Oscillator

In the 1920s, Balthasar Van der Pol experimented with some novel electrical circuits involving triodes. One of those circuits is now known as the Van der Pol oscillator, and is described by the differential equation

$$\ddot{x} = -x - \varepsilon(x^2 - 1)\dot{x}. \quad (5.59)$$

Interestingly, this equation pops up frequently, and is seen in the theory of the laser. If $\varepsilon = 0$, then this is a simple harmonic oscillator. But for $\varepsilon \neq 0$, this oscillator departs from the harmonic oscillator, and in a *nonlinear* way. Thus, many of the analytic methods normally applied to differential equations are not applicable to this problem!

If ε is small, then this problem can be treated by a form of perturbation theory, in which the effect of the nonlinear term is averaged over one cycle of the oscillation. But this is valid only if ε is small. Fortunately, our numerical methods *do not* rely on the linearity of the differential equation, and are perfectly happy to solve this equation for us.

EXERCISE 5.10

Use our friendly integrator to solve Equation (5.59), for $\varepsilon = 1$ (a value much greater than zero). As initial conditions, let $x(0) = 0.5$, and $\dot{x}(0) = 0$. Follow the solution for several oscillations, say, on the interval $0 \leq t \leq 8\pi$, and plot the results.

Phase Space

Although following the solution as a function of time is interesting, an alternate form of viewing the dynamics of a problem has also been developed. In a *phase space* description, the position and velocity (momentum) of a system are the quantities of interest. As time evolves, the point in phase space specifying the current state of the system changes, thereby tracing out a *phase trajectory*. For example, consider a simple harmonic oscillator, with

$$x(t) = \sin t \quad (5.60)$$

and

$$v(t) = \cos t \quad (5.61)$$

The position and velocity are plotted as functions of time in Figure 5.5. This is certainly a useful picture, but it's not the *only* picture. While the figure illustrates how position and velocity are (individually) related to time, it doesn't do a very good job of relating position and velocity to one another.

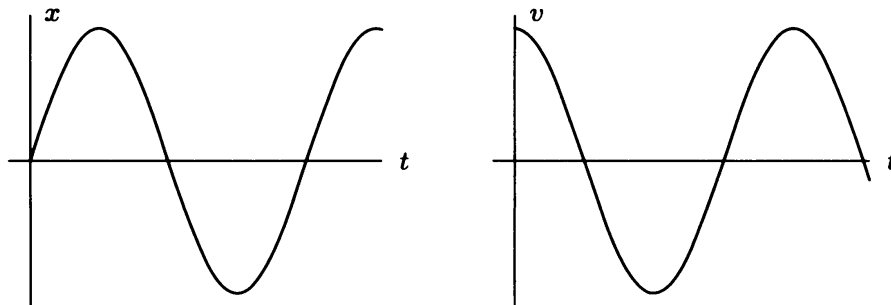


FIGURE 5.5 Position and velocity as functions of time for the simple harmonic oscillator.

But certainly position and velocity are related; we know that the total energy of the system is conserved. We can remove the explicit time dependence, thereby obtaining a direct relationship between them. Such a curve,

directly relating position and velocity, is shown in Figure 5.6. Of course, the time at which the system passes through a particular point can be noted, if desired.

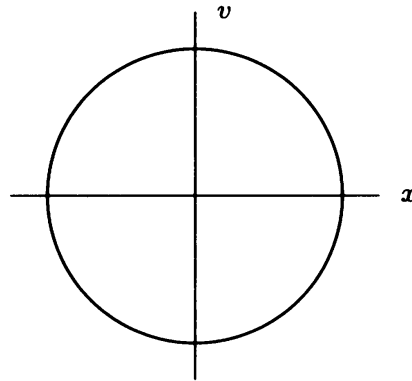


FIGURE 5.6 The trajectory in phase space of the simple harmonic oscillator.

By adopting a phase space picture, many results of advanced mechanics are immediately applicable. For example, the area enclosed by a phase trajectory is a conserved quantity for conservative systems, an oxymoron if ever there was one. For the simple harmonic oscillator, these phase trajectories are simple ellipses and can be denoted by the total energy, which is (of course) conserved.

Now consider a more realistic system, one with friction, for example, so that energy is dissipated. For the harmonic oscillator, the phase trajectory must spiral down and eventually come to rest at the origin, reflecting the loss of energy. This point is said to be an *attractor*.

We can also *drive* a system. That is, we can initially have a system at rest with zero energy, and *add* energy to it. A simple time-reversal argument should convince you that the phase trajectory will spiral outwards, evidencing an increase in the energy of the system.

But what if you have *both* dissipation and driving forces acting on a system? This is the case of the Van der Pol oscillator.

EXERCISE 5.11

Investigate the trajectory of the Van der Pol oscillator in phase space, for $0 \leq t \leq 8\pi$. Try different initial conditions, say with $\dot{x}(0) = 0$ but $x(0)$ ranging from 0.5 up to 3.0, or with $x(0)$ fixed and $\dot{x}(0)$ varying.

Plot some of your results, and write a paragraph or two describing them. From your investigations, can you infer the meaning of “limit cycle”?

The Finite Amplitude Pendulum

Back when we were talking about integrals in Chapter 4, we discussed the “simple” pendulum, whose motion is described by the differential equation

$$\ddot{\theta} = -\frac{g}{l} \sin \theta. \quad (5.62)$$

We found that the period of this pendulum was given in terms of an elliptic integral. We now want to solve for θ as a function of time. In the usual introductory treatments of the pendulum, $\sin \theta$ is approximated by θ and Equation (5.62) becomes the equation for a simple harmonic oscillator. In more advanced presentations, more terms in the expansion of the sine function might be retained and a series solution to Equation (5.62) be developed; but even then, the solution is valid only to the extent that the expansion for the sine is valid. But a numerical solution is not constrained in this way. With our handy-dandy second-order integrator up and running, it’s nearly trivial to generate accurate numerical solutions. In fact, we can generate several different solutions, corresponding to different initial conditions, to produce a family of trajectories in phase space, a *phase portrait* of the system. Such a portrait contains a lot of information. For example, at small amplitudes $\sin \theta$ is approximately θ , so that the path in phase space should look very similar to those for a simple harmonic oscillator. With larger amplitudes, the trajectories should look somewhat different.

Let’s recall some facts about the simple pendulum. The total energy, E , is the sum of kinetic and potential energies,

$$E = T + V, \quad (5.63)$$

where

$$T = \frac{1}{2} m l^2 \dot{\theta}^2 \quad (5.64)$$

and

$$V = m g l (1 - \cos \theta). \quad (5.65)$$

For $\theta_0 = 0$ we then have

$$E = T = \frac{1}{2} m l^2 \dot{\theta}_0^2 \quad (5.66)$$

or

$$\dot{\theta}_0 = \sqrt{\frac{2E}{ml^2}}. \quad (5.67)$$

With these initial conditions, Equation (5.62) can be solved to find θ and $\dot{\theta}$ as a function of time, for various total energies. Is there a physical significance to energies greater than $2mgl$?

EXERCISE 5.12

Solve the “simple” pendulum for arbitrary amplitudes. In particular, create a phase portrait of the physical system. For simplicity, choose units such that $g = l = 1$ and $m = 0.5$, and plot θ in the range $-3\pi < \theta < 3\pi$. Identify trajectories in phase space by their energy; include trajectories for $E = 0.25, 0.5, 0.75, 1, 1.25$, and 1.5 . (Why should you take particular care with the $E = 1$ trajectory? This particular trajectory is called the *separatrix* — I wonder why?)

The Animated Pendulum

As we noted, the phase space approach is achieved by removing the explicit time dependence. This yields valuable information, and presents it in a useful manner — the significance of an *attractor* is immediately obvious, for example. The presentation is *static*, however — quite acceptable for textbook publication, but not really in the “modern” spirit of things. With a microcomputer, you can generate the solution as a function of time — why don’t you display the solution as time evolves? In particular, why don’t you *animate* your display? Actually, it’s not that difficult!

Animation is achieved by displaying different images on the screen in quick succession. On television and at the cinema, the images are displayed at a rate of about 30 images per second. About 15 images a second is the lower limit; less than this, and the mind/eye sees “jitter” rather than “motion.” Affordable full-color, full-screen displays at 30 images per second are coming, but at the present time that is an expensive proposition; fortunately, what we need is not that challenging.

Imagine a “picture” of our pendulum, simply a line drawn from the point of suspension to the location of the pendulum bob. The next image is simply another line, drawn to the new location of the bob. For such simple line plots, we can achieve animation by simply erasing the old line and drawing the new one! Actually, we don’t erase the old one, we redraw it in the background color. Clearly, there are differences in the details of how this works

on a monochrome display, versus a color one. But we don't need to know the details; we just need to change the color. And that's done by the subroutine `color`, which we've used previously. The logic for the computer code might be something like the following:

```

        integer background, white, number
        ...
*   Initialize values
        x_fixed = ...
        y_fixed = ...
        time = ...
        theta = ...
        x_bob = ...
        y_bob =
        background = 0                !  BLACK background
        call noc( number )
        white = number - 1
        call color( white )
        call line(x_fixed,y_fixed,x_bob,y_bob)

100  < Start of time step>
        ...
*   < Propagate the solution one time step, >
*   < getting THETA at new time           >
        ...

*   < At this point, we have the new theta and >
*   < are ready to update the display. >

*   < Do we need to wait here? >
*
*   Erase old line
*
        call color( background )
        call line(x_fixed,y_fixed,x_bob,y_bob)
*
*   Get coordinates of bob from new THETA
*
        x_bob = ...
        y_bob = ...
*
*   Draw new line
*
```

```

        call color( white )
        call line(x_fixed,y_fixed,x_bob,y_bob)
        ...
*    < Do another time step?  or else?  >
        ...

```

Note that a line is displayed on the screen while a new θ is being calculated. Then, in quick succession, the old line is redrawn with the background color (hence erasing it) and the new line is drawn with the color set to white. For the animation to simulate the motion of the pendulum accurately, we would need to ensure that the lines are redrawn/drawn at a constant rate, perhaps by adding a timing loop to the code. But for purposes of illustration, it's sufficient to modify the code to use a fixed step size (in time). This time step should be small enough to maintain the accuracy of the calculation. If you find that the animation proceeds too rapidly, you can further decrease the step size and update the display only after several steps have been taken.

EXERCISE 5.13

Animate your pendulum.

Another Little Quantum Mechanics Problem

Way back in Chapter 2 we discussed the one-dimensional Schrödinger equation,

$$-\frac{\hbar^2}{2m} \frac{d^2\psi}{dx^2} + V(x) = E\psi(x), \quad (5.68)$$

and found solutions for the problem of an electron in a finite square well. Those solutions were facilitated by knowing the analytic solutions for the different regions of the potential. Well, that was then, and this is now!

Let's imagine a different physical problem, that of an electron bound in the anharmonic potential

$$V(x) = \alpha x^2 + \beta x^4, \quad (5.69)$$

where α and β are constants. In a qualitative sense, the solutions to this problem must be similar to those for the finite square well; that is, an oscillatory solution between the classical turning points, and a decaying solution as you move into the forbidden region. The sought-for solution should look something like Figure 5.7.

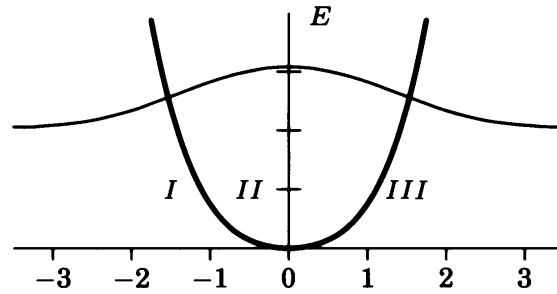


FIGURE 5.7 The anharmonic potential $V(x) = 0.5x^2 + 0.25x^4$, and its least energetic eigenfunction.

The computer code you presently have should work just fine, except for one small problem: what are the initial conditions? In this problem, you don't have them! Rather, you have *boundary conditions*. That is, instead of knowing ψ and $d\psi/dx$ at some specific point, you know how ψ should behave as $x \rightarrow \pm\infty$. That's a little different, and takes some getting used to.

There are various ways this difficulty can be overcome, or at least circumvented. For the immediate problem, we can simply *guess* initial conditions and observe their consequences. Deep in Region I, we know that ψ should be small but increasing as x increases; let's guess that

$$\psi(x_0) = 0,$$

and

$$\psi'(x_0) = \psi'_0, \quad (5.70)$$

where ψ'_0 is a positive number. We can then make an initial guess for the energy E , and “solve” the differential equation using the Runge–Kutta–Fehlberg integrator. Plotting the solution, we can see how good our guess for the energy was.

Before we can do the calculation, however, we need specific values for the constants. As noted earlier, the computer deals with pure numbers only, and not units, so that we need to exercise some care. Using an appropriate system of units is extremely advantageous. For example, the mass of the electron is 9.11×10^{-31} kilograms, but it's far more convenient, and less prone to error, to use a system of units in which the mass of the electron is defined as being 1. Using electron volts for energy and Angstroms for distance, we have

$$\hbar^2 = 7.6199682 m_e \text{ eV } \text{\AA}^2. \quad (5.71)$$

We'll take $\alpha = 0.5 \text{ eV } \text{\AA}^{-2}$, and $\beta = 0.25 \text{ eV } \text{\AA}^{-4}$. And to start the integration, we'll take $x_0 = -5 \text{ \AA}$, $\psi'_0 = 1 \times 10^{-5}$, and let $\text{EPSILON} = 1 \times 10^{-5}$. Figure 5.8

contains the results of such calculations, for trial energies of 1.5, 1.75, 2.0, and 2.25 electron volts. We need to discuss these results in some detail.

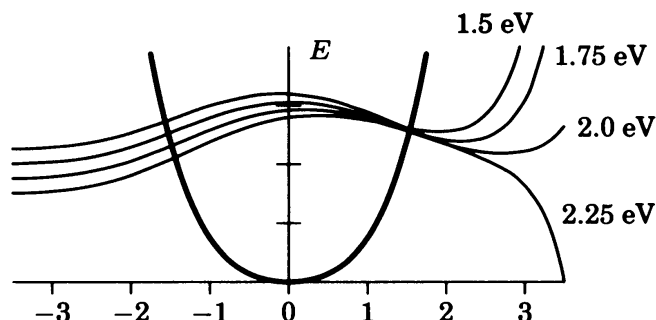


FIGURE 5.8 Numerical solutions of the anharmonic potential for trial energies of 1.5, 1.75, 2.0, and 2.25 electron volts. The numerical solution must satisfy the boundary conditions to be physically acceptable.

Consider the first trial energy attempted, 1.5 electron volts. In Region I, where the integration begins, the solution has the correct general shape, decreasing “exponentially” as it penetrates the barrier. In the classically allowed region the solution has a cosine-like behavior, as we would expect. But the behavior to the right of the well is clearly incorrect, with the solution becoming very large instead of very small. Recall that in this region there are two mathematically permitted solutions: one that is increasing, and one that is decreasing; it is on the basis of our *physical* understanding that only the decreasing one is permitted. But of course, our trial energy is not the correct energy, it’s only a first guess, and a rather poor one at that, as evidenced by the poor behavior of the solution. Recall that we want to find that energy which leads to a strictly decaying solution in this region.

Let’s try another energy, say, 1.75 electron volts. The behavior of the numerical solution is much as before, including a rapid increase to the right of the potential well. However, the onset of this rapid increase has been postponed until farther into the barrier, so that it is a better solution than we had previously. Let’s try again: at 2.0 electron volts, the onset is delayed even further. We’re coming closer and closer to the correct energy, finding solutions with smaller amounts of the increasing solution in them. So we try again, but as shown in the figure, we get a quite different behavior of the solution for 2.25 electron volts: instead of becoming very large and positive, it becomes very large and negative!

What has happened, of course, is that our trial energy has gotten too large. The correct energy, yielding a wavefunction that tends toward zero at

large x , is between 2 and 2.25 electron volts, and we see a drastic change in behavior as we go from below to above the correct energy. But can we ever expect to find that correct energy by this method? That is, can we ever find a solution that strictly decays in Region *III*? And why aren't we seeing any of this "explosion of the solution" *to the left* of the potential well?

Recall that there are two linearly independent solutions to a second-order differential equation; in the classically forbidden region, one of the solutions increases, and one decreases. Any solution can be written as a linear combination of these two: what we want is that *specific* solution which has the coefficient of the increasing solution exactly equal to zero. As the trial energy comes closer to the correct energy, that coefficient decreases — but we saw that our numerical solution always "blew up" at some point. Why? Although the coefficient is small, it's multiplying a function that is *increasing* as x increases — no matter how small the coefficient, as long as it is not *exactly* zero, there will come a point where this coefficient times the increasing function will be larger than the desired function, which is *decreasing*, and the numerical solution will "explode"! Why wasn't this behavior seen in Region *I*? In that region, the sought-after solution was *increasing*, not decreasing — the contribution from the nonphysical function was *decreasing* as the integration proceeded out of the classically forbidden region. In Region *I*, we integrated in the direction such that the unwanted contribution vanished, while in Region *III*, we integrated in the direction such that the unwanted solution *overwhelmed* our desired one. *We integrated in the wrong direction!*

Always propagate the numerical solution in the same direction as the physically meaningful solution increases.

The cure for our disease is obvious: we need always to begin the integration in a classically forbidden region, and integrate toward a classically allowed region. In the case of the potential well, we'll have two solutions, integrated from the left and right, and require that they match up properly in the middle. For symmetric potentials, such as the anharmonic potential we've been discussing, the situation is particularly simple since the solutions must be either even or odd: the even solutions have a zero derivative at $x = 0$, and the odd solutions must have the wavefunction zero at $x = 0$. It looks like we're almost ready to solve the problem.

But there are a couple of loose ends remaining. What about the choice of ψ'_0 ? Doesn't that make any difference at all? Well it does, but not much. With a different numerical value of ψ'_0 we would be lead to a different numerical solution, but one with exactly the same validity as the first, and only

differing from it by an overall multiplicative factor. For example, if $\psi(x)$ is a solution, so is $5\psi(x)$. This ambiguity can be removed by normalizing the wavefunction, e.g., requiring that

$$\int_{-\infty}^{\infty} \psi^*(x)\psi(x) dx = 1. \quad (5.72)$$

(We note that even then there remains an uncertainty in the overall phase of the wavefunction. That is, if ψ is a normalized solution, so is $i\psi$. For the bound state problem discussed here, however, the wavefunction can be taken to be real, and the complex conjugate indicated in the integral is unnecessary.) For our eigenvalue search, there's an even simpler approach. Since the *ratio* of ψ' to ψ eliminates any multiplicative factors, instead of searching for the zero of $\psi(0)$ we can search for the zero of the *logarithmic derivative*, and require

$$\left. \frac{\psi'(x, E)}{\psi(x, E)} \right|_{x=0} = 0 \quad (5.73)$$

for the eigenvalue E . (For the odd states, we'd want to find the zero of inverse of the logarithmic derivative, of course.)

Now, we're almost done; all that remains is to ensure that the calculation is accurate. If a (poor) choice of the tolerance yields 5 significant digits in the Runge–Kutta–Fehlberg integration, for example, then it's meaningless to try to find the root of the logarithmic derivative to 8 significant places. The overall accuracy can never be greater than the least accurate step in the calculation. We also need to verify that the x_0 is “deep enough” in the forbidden region that the eigenvalue doesn't depend upon its value.

EXERCISE 5.14

Find the lowest three eigenvalues, two even and one odd, of the anharmonic potential $V(x) = 0.5x^2 + 0.25x^4$, and plot the potential and the eigenfunctions. Discuss the measures you've taken to ensure 8-significant-digit accuracy in the eigenvalues.

We should not leave you with the impression that all the problems of quantum mechanics involve symmetric potentials — quite the contrary is true. Symmetry is a terrifically useful characteristic, and should be exploited whenever present. But the more usual situation is the one in which the potential is *not* symmetric. For example, let's consider the force between two atoms. When they are far apart, the electrons of one atom interact with the electrons and nucleus of the other atom, giving rise to an attractive force, the *van der Waals attraction*. But as the distance between the atoms becomes very small,

the force becomes repulsive as the nuclei (or the ionic core in many-electron atoms) interact with one another. Thus the general shape of the potential must be repulsive at small distances and attractive at large ones, necessitating an energy minimum somewhere in the middle. As an example, the potential energy curve for the hydrogen molecule is presented in Figure 5.9.

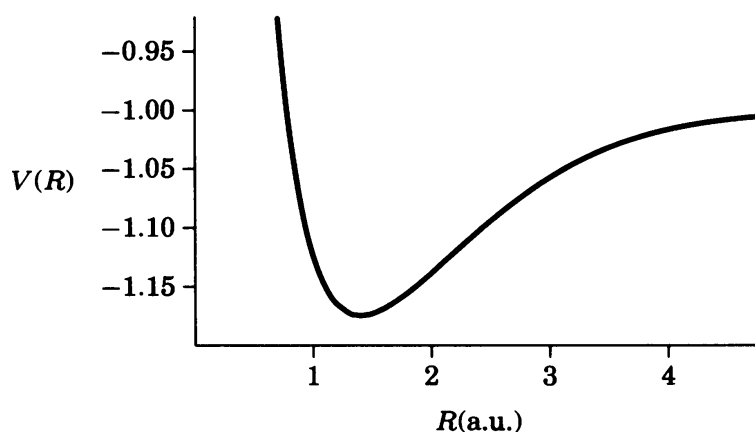


FIGURE 5.9 The ground state potential for molecular hydrogen. Energies are expressed in Hartree (≈ 27.2 eV) and distances in Bohr (≈ 0.529 Å). (Data taken from W. Kolos and L. Wolniewicz, “Potential-Energy Curves for the $X^1\Sigma_g^+$, $b^3\Sigma_u^+$, and $C^1\Pi_u$ States of the Hydrogen Molecule,” *Journal of Chemical Physics* **43**, 2429, 1965.)

Clearly, the potential is not symmetric. And since it *is not* symmetric, the eigenfunctions *are not* purely even or odd functions. Still, the method of solution is essentially the same as before: choose a *matching point*, say, near the minimum of the potential well. Then, beginning far to the left, integrate to the matching point; beginning far to the right, integrate to the matching point; and compare the logarithmic derivatives at the matching point.

Several Dependent Variables

We’ve seen that the solution of a second-order differential equation can be transformed into the solution of two, first-order differential equations. Regarding these equations as components of a vector, we were able to develop a computer code to solve the problem. This code can easily be extended in another direction, to problems having several dependent variables.

For example, to describe the motion of a particle in space we need the

particle's x , y , and z coordinates, which are all functions of time. The motion is described by Newton's second law, a second-order differential equation, so that we can introduce the variables v_x , v_y , and v_z to obtain a set of six dependent variables that fully describe the particle's motion. These variables can be regarded as the components of a six-dimensional vector — and we already know how to solve such vector problems! All that is required is to modify the computer code, changing the vectors and the indices on the loops from two to six.

As we've just seen, six variables are required *for each* particle — clearly, the number of independent variables in a problem is not simply the dimensionality of the space. But the algorithm we've developed, suitably modified to reflect the number of variables, can be applied to all such problems involving a single independent variable.

EXERCISE 5.15

Consider the problem of the falling sphere with air resistance, introduced in Exercise 5.5. In three dimensions, its motion is described by the vector equation

$$\frac{d\mathbf{p}}{dt} = m\mathbf{g} - kv^2 \frac{\mathbf{p}}{p}.$$

Note that \mathbf{p}/p is a unit vector pointing in the direction of the momentum. If the particle is initially at the origin traveling at 50 meters per second in the direction of the positive x -axis, what are the position and velocity of the sphere as a function of time, $0 < t < 10$ seconds? (How many dependent variables are required in this problem?)

Shoot the Moon

Newton tells us that the gravitational force of attraction between two bodies is

$$\mathbf{F} = -G \frac{mM}{r^2} \mathbf{e}_r. \quad (5.74)$$

We already know that $\mathbf{F} = m\mathbf{a}$, so I suppose we know all there is to know about the motion of bodies under the influence of gravity.

In principle, maybe; in practice, there's a little more to it. Let's start with a little review of a problem from freshman mechanics. Imagine that we have two bodies, say, the Earth and the Moon, that are orbiting one another. Or to put it another way, each body is orbiting about a common center-of-mass. This makes it convenient to use a coordinate system to describe the

system that is fixed in space and has its origin located at the center-of-mass. In general, the orbital trajectories are ellipses, with one focus at the origin, but for the Earth–Moon system the eccentricity of the orbit is so small, 0.055, that we'll approximate them as circles. If d is the (mean) center-to-center distance between the Earth and the Moon, then

$$r_E = \frac{m_M}{m_M + m_E} d \quad (5.75)$$

and

$$r_M = \frac{m_E}{m_M + m_E} d \quad (5.76)$$

where m_E is the mass of the Earth and m_M the mass of the Moon.

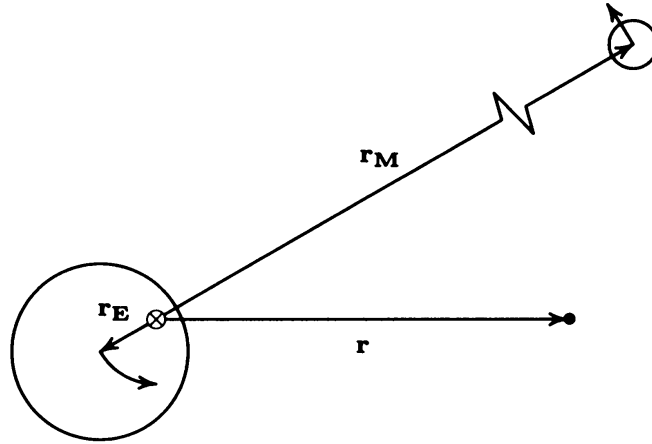


FIGURE 5.10 Center-of-mass coordinates for the Earth–Moon system. The view is from Polaris, looking down on the plane of rotation, with the counterclockwise motion of the Earth and Moon indicated. The relative sizes of the Earth and Moon, and the location of the center-of-mass \otimes , are drawn to scale.

The Earth–Moon system revolves about its common center of mass with a sidereal orbital period T , or with an angular frequency $\omega = 2\pi/T$. If the Moon lies on the positive x -axis at $t = 0$, then

$$\phi_M = \omega t \quad (5.77)$$

and

$$\phi_E = \omega t + \pi, \quad (5.78)$$

where ϕ_M is the angular location of the Moon and ϕ_E is the angular location of the Earth. The relationship between T , d , and the masses is not accidental, of course. To remain in uniform circular motion a body must be constantly

accelerated, the acceleration being generated by the gravitational force of attraction. Thus we have Kepler's third law of planetary motion, first published in *Harmonice Mundi* in 1619, that the square of the orbital period is proportional to the cube of the mean separation.

We now have a reasonable description of the Earth–Moon system. Let's add to the equation mankind's innate desire to explore the unknown, and ask: How do we journey from the Earth to the Moon?

Let's imagine that a spacecraft has been placed in a “parking orbit” 400 km above the Earth, so that the radius of the orbit is 6800 km. At time $t = 0$ the spacecraft reaches the angular position α in its circular orbit about the Earth, the rockets fire, and the spacecraft attains the velocity of v_0 in a direction tangent to its parking orbit. Since the rockets fire for only a brief time compared to the duration of the voyage, we'll assume it to be instantaneous. But of course, as the spacecraft moves towards the Moon, the Moon continues its revolution about the Earth–Moon center-of-mass. All the while, the motion of the spacecraft is dictated by Newton's laws,

$$\mathbf{F} = m\mathbf{a} = -G \frac{m m_E}{|\mathbf{r} - \mathbf{r}_E|^3} (\mathbf{r} - \mathbf{r}_E) - G \frac{m m_M}{|\mathbf{r} - \mathbf{r}_M|^3} (\mathbf{r} - \mathbf{r}_M). \quad (5.79)$$

As we had expected, the mass of the spacecraft doesn't enter into these equations. They can be written in component form as

$$\ddot{x} = -Gm_E \frac{x - x_e}{d_E^3} - Gm_M \frac{x - x_M}{d_M^3}, \quad (5.80)$$

and

$$\ddot{y} = -Gm_E \frac{y - y_e}{d_E^3} - Gm_M \frac{y - y_M}{d_M^3}, \quad (5.81)$$

where

$$d_E = \sqrt{(x - x_E)^2 + (y - y_E)^2} \quad (5.82)$$

is the distance from the spacecraft to the center of the Earth and

$$d_M = \sqrt{(x - x_M)^2 + (y - y_M)^2} \quad (5.83)$$

is the distance from the spacecraft to the center of the Moon. With knowledge of a few physical constants, such as

$$\begin{aligned} d &= 3.844 \times 10^5 \text{ km}, \\ m_E &= 5.9742 \times 10^{24} \text{ kg}, \\ m_M &= 7.35 \times 10^{22} \text{ kg}, \end{aligned}$$

$$T = 27.322 \text{ days,}$$

and

$$G = 6.6726 \times 10^{-11} \text{ N m}^2 / \text{kg}^2,$$

we're ready to explore!

EXERCISE 5.16

As a preliminary exercise, and to check that all the parameters have been correctly entered, set the mass of the Moon to zero and see if you can verify the freshman calculation for the velocity needed for the spacecraft to have a circular orbit about the Earth at this altitude.

Now, for the “real thing.”

EXERCISE 5.17

Find a set of initial conditions, α and v_0 , that will send the spacecraft from the parking orbit to the Moon. For starters, just try to hit the Moon. (Score a hit if you come within 3500 km of the center of the Moon.)

EXERCISE 5.18

For more intrepid voyagers, find initial conditions that will cause the spacecraft to loop around the Moon and return to Earth. This orbit is of real significance: see “Apollo 13” under the heading NEAR DIS-ASTERS in your history text.

Finite Differences

There is another approach to boundary value problems that's considerably different from the method we've been discussing. In it, we replace the given *differential* equation by the equivalent *difference* equation. Since we have approximations for derivatives in terms of finite differences, this shouldn't be too difficult. For example, consider the differential equation

$$y'' - 5y'(x) + 10y = 10x \tag{5.84}$$

subject to the boundary conditions

$$y(0) = 0, \quad y(1) = 100. \tag{5.85}$$

The first step is to impose a *grid* on this problem and to derive the appropriate finite difference equation. We will then seek the solution of the difference equation at the grid points. Since we are replacing the *continuous* differential equation by the *discrete* finite difference equation, we can only ask for the solution on the finite grid. We would hope, of course, that the solution to this problem is (at least approximately) a solution to the original one. For our example, we'll choose the grid to be $x = 0.0, 0.1, 0.2, \dots, 1.0$, and solve for the function on the interior points — the solutions at $x = 0$ and $x = 1$ are fixed by the boundary condition, and are not subject to change! To derive the finite difference equation, consider some arbitrary grid point, x_i ; at this point, we have (approximately)

$$y'_i \approx \frac{y_{i+1} - y_{i-1}}{2h}, \quad (5.86)$$

and

$$y''_i \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \quad (5.87)$$

where $h = x_{i+1} - x_i$ and we've introduced the notation $y_i = y(x_i)$. (Note that the expressions for the derivatives we're using are of the same order of accuracy, having error $O(h^2)$ — there would be no advantage to using an expression for one term in the differential equation that is more accurate than the expressions for any other term.) These approximations are substituted into the original differential equation to obtain the finite difference equation

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - 5\frac{y_{i+1} - y_{i-1}}{2h} + 10y_i = 10x_i. \quad (5.88)$$

This equation has the same structure as some of the equations we investigated in Chapter 2, and can be solved in the same manner. That is, we can write the equation in matrix form, and use Gaussian elimination to obtain a solution. However, there are other ways of solving such equations, which we should investigate. Rather than solving the problem in a *direct* manner, we'll develop an *indirect* one. Solving for y_i , we find

$$y_i = \frac{1}{2 - 10h^2} \left[\left(1 - \frac{5h}{2}\right)y_{i+1} + \left(1 + \frac{5h}{2}\right)y_{i-1} - 10h^2x_i \right]. \quad (5.89)$$

We will find an approximate solution to our differential equation by finding y 's that satisfy the derived difference equation, and we'll do that by *iteration*.

Iterative methods are usually not superior to direct methods when applied to ordinary differential equations. However, many physical situations are actually posed in several dimensions, and require the solution to partial differential equations rather than ordinary ones. In these instances, the relative merits are frequently reversed, with iterative methods being immensely

superior to direct ones. We should also note that with direct methods the error tends to accumulate as the solution is generated. In Gaussian elimination, for example, the back substitution propagates any error that may be present in one component to all subsequent components. Iterative methods, on the other hand, tend to treat all the components equally and distribute the error uniformly. An iterative solution can almost always be “improved” by iterating again — with a direct solution, what you get is all you got. In Chapter 7 we will explicitly discuss partial differential equations and appropriate methods to investigate them. But those methods are most easily introduced in the context of a single coordinate, e.g., ordinary differential equations, and hence we include their discussion at this point.

Let’s begin our development of an iterative method by simply guessing values for all the y ’s — for example, we can guess that y is a linear function and evaluate the y_i ’s accordingly. These become our *old* y ’s as we use Equation (5.89) to obtain *new* y ’s according to the iterative expression

$$y_i^{(j)} = \frac{1}{2 - 10h^2} \left[\left(1 - \frac{5h}{2}\right)y_{i+1}^{(j-1)} + \left(1 + \frac{5h}{2}\right)y_{i-1}^{(j-1)} - 10h^2 x_i \right]. \quad (5.90)$$

The initial guesses are denoted $y_i^{(0)}$, and might be stored in an array `Yold`. The first iteration at x_i is obtained from $y_{i+1}^{(0)}$ and $y_{i-1}^{(0)}$ by application of Equation (5.90), and stored in the array `Ynew`. One iteration consists of moving entirely through the array `Yold`, evaluating the elements of `Ynew` at all the interior points — remember, the first and last entries are fixed! After all the `Ynew` entries have been evaluated, `Ynew` can be copied into `Yold`, and one cycle of the iteration is complete; the process is then repeated. This is known as the *Jacobi* iteration scheme, and will converge to the correct result. However, we can speed it up a bit with no additional effort.

In the Jacobi scheme, the old values of y are used to evaluate the new ones. But think about that: you’ve just determined $y_i^{(j)}$, and are ready to evaluate $y_{i+1}^{(j)}$. Jacobi would have you use $y_i^{(j-1)}$ in this evaluation, although you’ve just calculated a better value! Let’s use the better value: in moving through the y -array from left to right, replace Equation (5.90) by the *Gauss–Siedel* iteration scheme

$$y_i^{(j)} = \frac{1}{2 - 10h^2} \left[\left(1 - \frac{5h}{2}\right)y_{i+1}^{(j-1)} + \left(1 + \frac{5h}{2}\right)y_{i-1}^{(j)} - 10h^2 x_i \right]. \quad (5.91)$$

Of course, if moving through the array from right to left, you would use

$$y_i^{(j)} = \frac{1}{2 - 10h^2} \left[\left(1 - \frac{5h}{2}\right)y_{i+1}^{(j)} + \left(1 + \frac{5h}{2}\right)y_{i-1}^{(j-1)} - 10h^2 x_i \right]. \quad (5.92)$$

Not only is this a more rapidly convergent scheme, it eliminates the need for two different arrays of data. The iteration is continued until a specified level of accuracy is obtained, for *all* points.

Previously we've discussed how the accuracy of a single quantity is determined; in the present case, we would require that the successive iterates $y_i^{(j-1)}$ and $y_i^{(j)}$ be the same to some specified number of significant digits. But here we need to require that this accuracy be met at *all* the grid points. We find it very convenient to use *logical variables* to do this, as suggested in the example computer code:

```
*
* This code fragment implements GAUSS-SIEDEL iteration to
* solve the finite difference Equation (5.91).
*
      Double Precision y(11), h, c1,c2,c3,c4, x, yy
      +           , tolerance
      Integer i, iteration
      LOGICAL DONE
      Parameter ( tolerance = 5.d-4 )
      ...
      h = ...
      c1 = 1.d0 - 2.5d0 * h
      c2 = 1.d0 + 2.5d0 * h
      c3 = -10.d0 * h * h
      c4 = 2.d0 + c3
*
* Initialize y:
*
      DO i = 1, 11
         y(i) = ...
      END DO
      iteration = 0
*
* Iterate until done...or have iterated too many times.
*
100    DONE = .TRUE.
        iteration = iteration + 1
        IF (iteration .ge. 100) stop 'Too many iterations!'
*
* Evaluate the function at all the interior points:
*
      DO i = 2, 10
         x = ...
```

```

        yy = (c1 * y(i+1) + c2 * y(i-1) + c3 * x ) / c4
        IF ( abs( (yy-y(i))/yy ) .gt. tolerance )
+           DONE = .FALSE.
        y(i) = yy
    END DO
    IF (.NOT.DONE)goto 100
    ...

```

The variable `DONE` is declared to be a logical variable, and is set to `TRUE` at the start of every iteration. As the iteration proceeds, the accuracy of each point is tested. Any time the error exceeds the specified accuracy, `DONE` is set to `FALSE` — of course, it only takes one such instance for `DONE` to become `FALSE`. The final accuracy check is then very simple, “if not done, iterate again.” By choosing the appropriate type of variable, and a suitable name for it, the convergence checking has been made very clear.

The accuracy we’ve specified is not great, for several reasons. First, it’s always wise to be conservative when starting a problem — try to develop some feel for the problem and the method of solution before you turn the computer loose on it. And secondly, we shouldn’t lose sight of the fact that this is a derived problem, not the original differential equation. Is an exact solution to an approximate problem any better than an approximate solution to the exact problem? To solve the difference equation to higher accuracy is unwarranted, since the difference equation is only a finite representation of the differential equation given. As a final comment on the computer code, note that, as should be done with any iterative method, the number of iterations are counted and a graceful exit is provided if convergence is not obtained. The maximum iteration count is set rather large; the method is guaranteed to converge, but it can be slow. The code fragment was used as a base to develop a computer program to solve the finite difference equation, with the following results:

iter	y(0.0)	y(0.1)	y(0.2)	y(0.3)	y(0.4)	y(0.5)	y(0.6)	y(0.7)	y(0.8)	y(0.9)	y(1.0)
0	0.00	10.00	20.00	30.00	40.00	50.00	60.00	70.00	80.00	90.00	100.00
1	0.00	7.89	17.02	26.97	37.46	48.30	59.38	70.61	81.94	93.33	100.00
2	0.00	6.71	15.05	24.68	35.28	46.62	58.51	70.80	83.38	94.28	100.00
3	0.00	5.94	13.64	22.88	33.44	45.07	57.57	70.75	83.72	94.50	100.00
4	0.00	5.38	12.56	21.45	31.88	43.67	56.63	70.26	83.49	94.35	100.00
5	0.00	4.95	11.71	20.27	30.55	42.43	55.62	69.51	82.93	93.99	100.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
86	0.00	1.98	5.04	9.48	15.65	23.90	34.52	47.68	63.34	81.10	100.00

Although the calculation has converged, note that 86 iterations were necessary to obtain the specified accuracy.

EXERCISE 5.19

Write a computer program to reproduce these results.

SOR

This whole method is referred to as *relaxation* — the finite difference equations are set up, and the solution *relaxes* to its correct value. As with any iterative procedure, the better the initial guess, the faster the method will converge. Once a “sufficiently accurate” approximation is available, either through good initialization or after cycling through a few iterations, the relaxation is monotonic — that is, each iteration takes the approximate solution a step closer to its converged limit, each step being a little less than the previous one. (In the table above, we see that the relaxation becomes monotonic after the third iteration.) This suggests an improved method, in which the *change* in the function from one iteration to the next is used to generate a better approximation: let’s *guess* that the converged result is equal to the most recent iterate plus some multiplicative factor times the difference between the two most recent iterations. That is, we guess the solution $\bar{y}^{(j)}$ as being

$$\bar{y}_i^{(j)} = y_i^{(j)} + \alpha \left[y_i^{(j)} - y_i^{(j-1)} \right], \quad (5.93)$$

where α lies between 0 and 1. (The optimal value for α depends on the exact structure of the difference equation.) This is called *over-relaxation*; since it’s repeated at each iteration, the method is referred to as *Successive Over-Relaxation (SOR)*.

EXERCISE 5.20

Modify your program to use the SOR method. Experiment with a few different values of α to find the one requiring the fewest iterations. For that α , you should see a dramatic improvement in the rate of convergence over the Gauss–Siedel method.

Discretisation Error

We noted earlier that demanding excessive accuracy in the solution of the finite difference equations is not appropriate. The reason, of course, is that the finite difference equation is only an *approximation* to the differential equation that we want to solve. Recall that the finite difference equation was obtained

by approximating the derivative on a grid — no amount of effort spent on solving the finite difference equation itself will overcome the error incurred in making that approximation. Of course, we could try a different grid

To make our point, we've solved the finite difference equation for three different grids, with $h = 0.2, 0.1$, and 0.05 . We pushed the calculation to 8 significant digits, far more than is actually appropriate, to demonstrate that the error is associated with the finite difference equation itself, and not the method of solution. That is, our results are essentially *exact*, for each particular grid used. Any differences in our results for different grids are due to the intrinsic error associated with the discretisation of the grid! These results are presented in Table 5.1.

TABLE 5.1 Discretisation Error

	$y(0.2)$	$y(0.4)$	$y(0.6)$	$y(0.8)$
<i>Finite difference</i>				
$h = 0.20$	4.3019	13.9258	31.9767	61.0280
$h = 0.10$	5.0037	15.5634	34.3741	63.2003
$h = 0.05$	5.1586	15.9131	34.8676	63.6292
<i>Richardson extrapolation</i>				
0.20/0.10	5.2376	16.0193	35.1732	63.9244
0.10/0.05	5.2103	16.0296	35.0321	63.7721
0.20/0.10/.05	5.2085	16.0243	35.0227	63.7619
<i>Analytic result</i>	5.2088	16.0253	35.0247	63.7644

As expected, the results become more accurate as h is decreased. But the *magnitude* of the discretisation error is surprising: at $x = 0.2$, for example, the calculated value changes by 16% as the grid decreases from $h = 0.2$ to 0.1 , and another 3% when h is further reduced to 0.05 . Again, these changes are due to the discretisation of the grid itself, not the accuracy of the solution of the finite difference equations. Clearly, it is a serious mistake to ask for many-significant-digit accuracy with a coarse grid — the results are not *relevant* to the actual problem you're trying to solve!

Since the error is due to the approximation of the derivatives, which we know to be $O(h^2)$, we can use Richardson's extrapolation to better our results. Using just the $h = 0.2$ and 0.1 data yields extrapolated results superior to the $h = 0.05$ ones. The extrapolations can themselves be extrapolated, with remarkable success.

Another approach to generating more accurate results would be to use

a more accurate approximation for the derivatives in the first place. By using a higher order approximation, the truncation error incurred by discarding the higher terms in the derivative expressions would be reduced. For the interior points, we can use the approximations developed in Chapter 3,

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + O(h^4) \quad (5.94)$$

and

$$f''(x) = \frac{-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)}{12h^2} + O(h^4). \quad (5.95)$$

These expressions have error $O(h^4)$, but are not applicable for the points immediately adjoining the endpoints, since either $f(x-2h)$ or $f(x+2h)$ will lie outside the range being considered. For these points, we must devise alternate expressions for the derivatives.

In Chapter 3, we used Taylor series expansions for $f(x+h)$, $f(x+2h)$, etc., to develop approximations for the derivatives. Let's try that again — the relevant expansions are

$$\begin{aligned} f(x-h) &= f_{-1} = f_0 - hf'_0 + \frac{h^2}{2!}f''_0 - \frac{h^3}{3!}f'''_0 + \frac{h^4}{4!}f^{iv}_0 - \frac{h^5}{5!}f^v_0 + O(h^6), \\ f(x) &= f_0, \\ f(x+h) &= f_1 = f_0 + hf'_0 + \frac{h^2}{2!}f''_0 + \frac{h^3}{3!}f'''_0 + \frac{h^4}{4!}f^{iv}_0 + \frac{h^5}{5!}f^v_0 + O(h^6), \\ f(x+2h) &= f_2 = f_0 + 2hf'_0 + \frac{4h^2}{2!}f''_0 + \frac{8h^3}{3!}f'''_0 + \frac{16h^4}{4!}f^{iv}_0 + \frac{32h^5}{5!}f^v_0 + O(h^6), \\ f(x+3h) &= f_3 = f_0 + 3hf'_0 + \frac{9h^2}{2!}f''_0 + \frac{27h^3}{3!}f'''_0 + \frac{81h^4}{4!}f^{iv}_0 + \frac{243h^5}{5!}f^v_0 + O(h^6), \\ f(x+4h) &= f_4 = f_0 + 4hf'_0 + \frac{16h^2}{2!}f''_0 + \frac{64h^3}{3!}f'''_0 + \frac{256h^4}{4!}f^{iv}_0 + \frac{1024h^5}{5!}f^v_0 \\ &\quad + O(h^6). \end{aligned} \quad (5.96)$$

Since we are seeking a more accurate approximation, it's necessary that we retain more terms in the expansion than before. And since the derivative is desired at a nonsymmetric location, we'll need to evaluate the function at

more points as well. Taking a linear combination of these expressions, we find

$$\begin{aligned}
& a_{-1}f_{-1} + a_0f_0 + a_1f_1 + a_2f_2 + a_3f_3 + a_4f_4 \\
&= f_0 [a_{-1} + a_0 + a_1 + a_2 + a_3 + a_4] \\
&+ hf'_0 [-a_{-1} + a_1 + 2a_2 + 3a_3 + 4a_4] \\
&+ \frac{h^2}{2!} f''_0 [a_{-1} + a_1 + 4a_2 + 9a_3 + 16a_4] \\
&+ \frac{h^3}{3!} f'''_0 [-a_{-1} + a_1 + 8a_2 + 27a_3 + 64a_4] \\
&+ \frac{h^4}{4!} f^{iv}_0 [a_{-1} + a_1 + 16a_2 + 81a_3 + 256a_4] \\
&+ \frac{h^5}{5!} f^v_0 [a_{-1} + a_1 + 32a_2 + 243a_3 + 1024a_4] \\
&+ O(h^6). \tag{5.97}
\end{aligned}$$

To obtain an expression for f'_0 , we require that the coefficients of the higher order derivatives vanish. To have an error of $O(h^4)$, it's sufficient to set $a_4 = 0$ and to choose the remaining coefficients such that

$$\begin{aligned}
& a_{-1} + a_1 + 4a_2 + 9a_3 = 0, \\
& -a_{-1} + a_1 + 8a_2 + 27a_3 = 0, \\
& a_{-1} + a_1 + 16a_2 + 81a_3 = 0. \tag{5.98}
\end{aligned}$$

With $a_{-1} = -3a_3$, $a_1 = 18a_3$, and $a_2 = -6a_3$, we find

$$f'_0 = \frac{-3f_{-1} - 12f_0 + 18f_1 - 6f_2 + f_3}{12h} + O(h^4). \tag{5.99}$$

Similarly, we can — with some effort — find that

$$f''_0 = \frac{10f_{-1} - 15f_0 - 4f_1 + 14f_2 - 6f_3 + f_4}{12h^2} + O(h^4). \tag{5.100}$$

As you can see, this approach leads to a rather more complicated scheme. And we have yet to derive the finite difference equations themselves. With these expressions for the derivatives, however, the difference equations are within sight. Note that there will be three distinct cases to be treated: 1) an endpoint, which is held constant; 2) a point adjacent to an endpoint, for which Equations (5.99) and (5.100) are used to derive the difference equation; and 3) points that are at least one point removed from an endpoint, for which the central difference approximations can be used to derive the appropriate difference equation.

EXERCISE 5.21

Modify your existing program to use these higher order approximations. Note that the optimal acceleration parameter determined previously will not be optimal for this new situation, although it should be a reasonable value to use.

Every physical situation is unique, but the general consensus is that the additional effort required for the higher order approximations is usually not justified. On a coarse grid, they perform no better than the three-point rule, and for fine grids the extra level of accuracy can be more easily obtained by using Richardson extrapolation to “polish” the coarse results.

A Vibrating String

You probably recall that the vibration of a string, fixed at both ends and under uniform tension, is described by the differential equation

$$\frac{\partial^2 u(x, t)}{\partial t^2} = \frac{T}{\mu(x)} \frac{\partial^2 u(x, t)}{\partial x^2}, \quad (5.101)$$

where T is the tension in the string and $\mu(x)$ is the mass of the string per unit length. A standard way to solve this problem is *to guess a solution* of the form

$$u(x, t) = y(x)\tau(t) \quad (5.102)$$

and see if it works! After making the substitution and rearranging, we find that

$$\frac{1}{y(x)} \frac{T}{\mu(x)} \frac{d^2 y}{dx^2} = \frac{1}{\tau(t)} \frac{d^2 \tau}{dt^2}. \quad (5.103)$$

Now, the left side of this equation is a function of x alone, and the right side a function of t alone. The only way this relation can be valid for all x and t is for both sides of the equation to equal a constant, which we take to be $-\omega^2$. We then have the *two* equations

$$\frac{1}{\tau(t)} \frac{d^2 \tau}{dt^2} = -\omega^2 \quad (5.104)$$

and

$$\frac{1}{y(x)} \frac{T}{\mu(x)} \frac{d^2 y}{dx^2} = -\omega^2. \quad (5.105)$$

Multiplying Equation (5.104) by $\tau(t)$ we find the differential equation

$$\frac{d^2\tau}{dt^2} + \omega^2\tau = 0, \quad (5.106)$$

with solution

$$\tau(t) = a \sin \omega t + b \cos \omega t. \quad (5.107)$$

The constant ω , which we introduced in order to separate our two equations, is an angular frequency with units of radians/second, and is related to the simple frequency ν , having units cycles/second, by

$$\omega = 2\pi\nu. \quad (5.108)$$

In like manner, Equation (5.105) leads us to the equation

$$\frac{T}{\mu(x)} \frac{d^2y(x)}{dx^2} + \omega^2y(x) = 0. \quad (5.109)$$

This looks similar to the problems we've solved earlier, but with a crucial difference: *we don't know ω !*

For the moment, let's assume that μ is a constant, μ_o . Then the differential equation (5.109) can be written as

$$\frac{d^2y(x)}{dx^2} + \frac{\omega^2\mu_o}{T}y(x) = 0. \quad (5.110)$$

We recognize this equation, and know that its general solution is simply

$$y(x) = \alpha \sin \omega \sqrt{\frac{\mu_o}{T}}x + \beta \cos \omega \sqrt{\frac{\mu_o}{T}}x. \quad (5.111)$$

Now we impose the boundary conditions: the string is fixed at each end, i.e., $y(x) = 0$ at $x = 0$ and $x = L$. For $y(x)$ to be zero at $x = 0$, we must have $\beta = 0$. And to insure that $y(L) = 0$, we must have

$$\sin \omega \sqrt{\frac{\mu_o}{T}}L = 0 \quad (5.112)$$

or

$$\omega \sqrt{\frac{\mu_o}{T}}L = n\pi, \quad n = 1, 2, \dots \quad (5.113)$$

Thus only certain values of ω are possible, those for which

$$\omega = \frac{n\pi}{L} \sqrt{\frac{T}{\mu}}, \quad n = 1, 2, \dots \quad (5.114)$$

This is the expected result — a string vibrates at only certain frequencies, determined by the string's mass density μ , length L , and tension T .

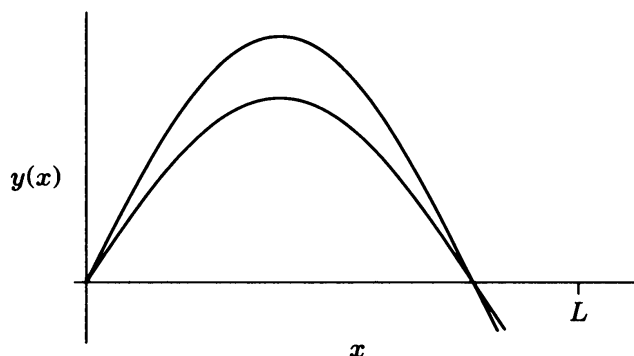


FIGURE 5.11 Two attempts at a numerical solution to the vibrating string problem, with different initial derivatives $y'(0)$. Neither satisfies the boundary condition at $x = L$.

Imagine, for a moment, that we had not recognized the analytic solution to Equation (5.110). Could we have tried to solve it numerically? We would very likely have tried the same method as we used for finding the eigenvalues of the anharmonic quantum oscillator, and it would work! We would first guess a k , and then guess an initial derivative $y'(0)$. That would give us sufficient information to generate a numerical solution on the interval $0 \leq x \leq L$. Then we might guess a different $y'(0)$, and generate another numerical solution. But as seen in Figure 5.11, this second guess wouldn't contribute to our finding the eigenvalue k . If y is a solution to the differential equation of Equation (5.110), then so is any multiple of y . Choosing a different initial derivative does nothing more than to select a different overall multiplicative factor. The eigenvalue can be determined, however, by searching through k to yield a numerical solution that is zero at $x = L$.

EXERCISE 5.22

Consider a piano wire 1 meter long, of mass 0.954 grams, stretched with a tension of 1000 Newtons. Use the *shooting method* we've described to find the lowest eigenvalue of the vibrating string problem, e.g., the frequency of its fundamental note.

This particular example is useful for purposes of illustration, but you already knew the answer. Consider a slightly different problem, one for which $\mu(x)$ is *not constant*. For example, the string might increase in thickness as you move along its length, so that its mass be described by the expression

$$\mu(x) = \mu_0 + (x - \frac{L}{2})\Delta. \quad (5.115)$$

μ_0 is the average mass density, and Δ is its variation per unit length.

EXERCISE 5.23

Reconsider the piano wire problem, with $\mu_0 = 0.954$ g/m and $\Delta = 0.5$ g/m². By how much does the fundamental frequency change, compared to the uniform string? Plot the shape of the string, $y(x)$, and compare to the sinusoidal shape of the string in the previous exercise.

Eigenvalues via Finite Differences

For problems involving one dimension, the shooting method is nearly always adequate in finding the solution to boundary value problems. However, the method is severely limited in two or more dimensions. The reason is simply that in one dimension the boundary condition is at a *point*, while in two dimensions, for example, the boundary is along a *curve*, i.e., an *infinite number of points*. If we are to solve multidimensional problems, then we must have an alternate method of solution. Such an alternate is obtained from the finite difference approach. In Chapter 7 this approach will be applied to two-dimensional problems; the presentation here will be in one dimension, to demonstrate how it proceeds.

Let's consider the nonuniform vibrating string problem we've investigated earlier. Replacing the derivatives in Equation (5.109) by their finite difference approximations, we obtain the difference equations

$$\frac{T}{\mu_i} \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} + \omega^2 f_i = 0, \quad i = 1, 2, \dots, N-1. \quad (5.116)$$

We can rewrite these equations as a single matrix equation, of the form

$$\begin{bmatrix}
-2\frac{T}{\mu_1 h^2} & \frac{T}{\mu_1 h^2} & & & \\
\frac{T}{\mu_2 h^2} & -2\frac{T}{\mu_2 h^2} & \frac{T}{\mu_2 h^2} & & \\
& \ddots & \ddots & \ddots & \\
& & \frac{T}{\mu_{N-1} h^2} & -2\frac{T}{\mu_{N-1} h^2} & \frac{T}{\mu_{N-1} h^2} \\
& & & \frac{T}{\mu_N h^2} & -2\frac{T}{\mu_N h^2}
\end{bmatrix}
\begin{bmatrix}
f_1 \\
f_2 \\
\vdots \\
f_{N-1} \\
f_N
\end{bmatrix}
= \omega^2
\begin{bmatrix}
f_1 \\
f_2 \\
\vdots \\
f_{N-1} \\
f_N
\end{bmatrix}. \quad (5.117)$$

Using matrix notation, we can write this as

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}, \quad (5.118)$$

which we recognize as the matrix eigenvalue equation with eigenvalue $\lambda = \omega^2$. Eigenvalue equations are quite common in physics, appearing frequently and in many different contexts. Moving the right side of the equation to the left, we have

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = 0, \quad (5.119)$$

where \mathbf{I} is the identity matrix. Written in this form, we can now bring all the power of linear algebra to bear upon the problem. In particular, we know that the only way for this equation to have a solution, other than the trivial one $\mathbf{x} = 0$, is for the determinant of the coefficient matrix to be zero,

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0. \quad (5.120)$$

This is the *secular equation*. Those values of λ for which the determinant vanishes are the *eigenvalues* of the matrix \mathbf{A} .

For an arbitrary matrix \mathbf{A} , finding the eigenvalues can be a formidable task. When expanded, the determinant of an $N \times N$ matrix will be an N -th order polynomial in λ . For the example we've been investigating, however, the

matrix A is tridiagonal and hence much simpler than the general problem. Let's write the matrix as

$$A - \lambda I = \begin{bmatrix} b_1 - \lambda & c_1 & & & \\ a_2 & b_2 - \lambda & c_2 & & \\ & a_3 & b_3 - \lambda & c_3 & \\ & & \ddots & \ddots & \ddots \\ & & & a_{N-1} & b_{N-1} - \lambda & c_{N-1} \\ & & & & a_N & b_N - \lambda \end{bmatrix}. \quad (5.121)$$

Because of the unique structure of the tridiagonal system, we can easily evaluate its determinant. Imagine that $A - \lambda I$ were a 1×1 matrix. Then its determinant, D_1 , would simply be $b_1 - \lambda$. If it were 2×2 , then

$$D_2 = (b_2 - \lambda)D_1 - a_2c_1, \quad (5.122)$$

and if 3×3 ,

$$D_3 = (b_3 - \lambda)D_2 - a_3c_2D_1. \quad (5.123)$$

In general, we have the recurrence relation

$$D_j = (b_j - \lambda)D_{j-1} - a_jc_{j-1}D_{j-2}, \quad j = 3, \dots, N. \quad (5.124)$$

After evaluating D_1 and D_2 , Equation (5.124) can be used to generate all the determinants up to N , which is the determinant we want. If we combine a root search with the evaluation of the determinant, we obtain an eigenvalue solver for the tridiagonal system. An outline of the program might look like the following:

```
* Initialize a, b, and c.
*
  DO i = 1, N
    x = ..
    mu = mu0 + (x-L/2)*Delta
    a(i) = T/(h*h*mu)
    b(i) = -2 * a(i)
    c(i) = a(i)
  END DO

  call TriEigen( a, b, c, lambda )

*
* We actually want the frequency, which is the
* square root of the eigenvalue lambda.
*
```

```

        omega = sqrt ( lambda )
        ...
    end

*-----
    Subroutine TriEigen( a, b, c, BestLambda )
    double precision a(11), b(11), c(11), BestLambda
*
* This subroutine searches DET, as a function of LAMBDA,
* for its first zero. One of the root-finding routines
* from Chapter 2 should be used here.
*
        ...

    end

*-----
    Double Precision Function Det(lambda,n)
    double precision x,a(11),b(11),c(11)
    DET = b(1) - lambda
    IF(N .ne. 1) THEN
        d1 = det
        DET = (b(2)-lambda) * d1 - a(2)*c(1)
        IF(N .ne. 2) THEN
            d2 = det
*
            DO j = 3, n
                DET = (b(j)-lambda)*d2-a(j)*c(j-1)*d1
                d1 = d2
                d2 = det
            END DO
        ENDIF
    ENDIF
*
    end

```

EXERCISE 5.24

Fill in the missing pieces in this program, and use it to determine the fundamental frequency of vibration of the nonuniform piano wire.

The Power Method

Since an N -th order polynomial has N roots, there are N eigenvalues to an $N \times$

N matrix. And with each of these eigenvalues λ_j is associated an eigenvector \mathbf{u}_j , e.g.,

$$\mathbf{A}\mathbf{u}_j = \lambda_j\mathbf{u}_j, \quad j = 1, 2, \dots, N. \quad (5.125)$$

One of the simplest ways of finding eigenvalues and eigenvectors of a general matrix, at least conceptually, is the so-called *power method*. Let's begin with some arbitrary vector \mathbf{x} — such a vector can be written as a linear combination of eigenvectors,

$$\mathbf{x} = \sum_{j=1}^N a_j \mathbf{u}_j. \quad (5.126)$$

For convenience, we'll assume that the eigenvalues are ordered such that

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_N|. \quad (5.127)$$

Now, let the matrix \mathbf{A} multiply \mathbf{x} , with the result that

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{A} \sum_{j=1}^N a_j \mathbf{u}_j \\ &= \sum_{j=1}^N a_j \mathbf{A}\mathbf{u}_j \\ &= \sum_{j=1}^N a_j \lambda_j \mathbf{u}_j. \end{aligned} \quad (5.128)$$

Multiply both sides by \mathbf{A} again, and then again, so that after m multiplications we have

$$\begin{aligned} \mathbf{A}^m \mathbf{x} &= \sum_{j=1}^N a_j \lambda_j^m \mathbf{u}_j \\ &= \lambda_1^m \left[a_1 \mathbf{u}_1 + \sum_{j=2}^N a_j \left(\frac{\lambda_j}{\lambda_1} \right)^m \mathbf{u}_j \right]. \end{aligned} \quad (5.129)$$

Since λ_1 is the dominant (largest magnitude) eigenvalue, the magnitude of the ratio λ_j/λ_1 is less than 1. As $m \rightarrow \infty$ this ratio tends to zero, leaving only the first term,

$$\lim_{m \rightarrow \infty} \mathbf{A}^m \mathbf{x} = \lambda_1^m a_1 \mathbf{u}_1. \quad (5.130)$$

If we have some vector \mathbf{y} — and any vector that has some component of \mathbf{u}_1 , the dominant eigenvector, will do — we can compute the scalar product $\mathbf{y}^T \mathbf{A}^m \mathbf{x}$.

Taking the ratio of consecutive scalar products, we have

$$\lim_{m \rightarrow \infty} \frac{\mathbf{y}^T \mathbf{A}^{m+1} \mathbf{x}}{\mathbf{y}^T \mathbf{A}^m \mathbf{x}} = \lambda_1. \quad (5.131)$$

EXERCISE 5.25

Use the power method to find the dominant eigenvalue of the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.132)$$

Almost any initial vector will work, some better than others. You might try

$$\mathbf{x} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \quad (5.133)$$

You can use $\mathbf{y} = \mathbf{x}$ to approximate λ_1 .

Eigenvectors

In many situations the eigenvalues — such as the frequencies of a vibration — are the primary quantities of interest. In these cases, we don't bother finding the eigenvector \mathbf{x} . (Note that in the shooting method, this is not an option; the eigenvector — the approximate $y(x)$ — is always generated.) But in other cases, we might want to find the eigenvector. In the nonuniform piano wire problem, for example, it's important to discover that the solution is *not* simply a sine curve. This information is contained in the eigenvector.

Let's return to the discussion of the power method, and let \mathbf{x} be a *normalized* vector so that

$$\mathbf{x}^T \mathbf{x} = 1. \quad (5.134)$$

Being normalized is not a requirement to being an eigenvector, but it's a desirable characteristic in that it can substantially simplify the analysis. Let's define a sequence of such normalized vectors, beginning with

$$\mathbf{x}^{(0)} = \mathbf{x}. \quad (5.135)$$

We then consider the product $\mathbf{A}\mathbf{x}^{(0)}$. The result of this multiplication will be a vector, but it will not be normalized. Let's normalize this vector, and call it

$\mathbf{x}^{(1)}$,

$$\mathbf{x}^{(1)} = \frac{\mathbf{A}\mathbf{x}^{(0)}}{\sqrt{|\mathbf{A}\mathbf{x}^{(0)}|^2}}. \quad (5.136)$$

With this definition of $\mathbf{x}^{(1)}$, we thus have

$$\mathbf{A}\mathbf{x}^{(0)} = q^{(1)}\mathbf{x}^{(1)} \quad (5.137)$$

where

$$q^{(1)} = \sqrt{|\mathbf{A}\mathbf{x}^{(0)}|^2}. \quad (5.138)$$

We then consider $\mathbf{x}^{(2)}$ as the normalized vector resulting from the product $\mathbf{A}\mathbf{x}^{(1)}$, so that

$$\mathbf{A}\mathbf{x}^{(1)} = q^{(2)}\mathbf{x}^{(2)}, \quad (5.139)$$

and so on. The sequence of $q^{(m)}$ and $\mathbf{x}^{(m)}$ thus constructed tends toward the dominant eigenvalue and its eigenvector,

$$\lim_{m \rightarrow \infty} q^{(m)} = \lambda_1, \quad \text{and} \quad \lim_{m \rightarrow \infty} \mathbf{x}^{(m)} = \mathbf{u}_1. \quad (5.140)$$

EXERCISE 5.26

Find the normalized eigenvector associated with the dominant eigenvalue of the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.141)$$

To find the eigenvalue of smallest magnitude, we consider the *inverse power method*. If

$$\mathbf{A}\mathbf{u}_j = \lambda_j\mathbf{u}_j, \quad (5.142)$$

then we can multiply by $\lambda_j^{-1}\mathbf{A}^{-1}$ to find that

$$\mathbf{A}^{-1}\mathbf{u}_j = \lambda_j^{-1}\mathbf{u}_j. \quad (5.143)$$

That is, if \mathbf{A} has eigenvalues λ_j and eigenvectors \mathbf{u}_j , the inverse of \mathbf{A} has the same eigenvectors \mathbf{u}_j but eigenvalues λ_j^{-1} . Then the eigenvalue of \mathbf{A} having the smallest magnitude will be the dominant eigenvalue of \mathbf{A}^{-1} ! To find that eigenvalue we use the same iteration scheme as before, but with \mathbf{A}^{-1} in place of \mathbf{A} ,

$$[\mathbf{A}^{-1}]\mathbf{x}^{(m)} = q^{(m+1)}\mathbf{x}^{(m+1)}. \quad (5.144)$$

EXERCISE 5.27

Verify that

$$\mathbf{A}^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} \quad (5.145)$$

is the inverse of

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}. \quad (5.146)$$

Then calculate the eigenvalue of \mathbf{A} with the smallest magnitude by using the power method to determine the dominant eigenvalue of \mathbf{A}^{-1} .

Rather than using the inverse explicitly, we can implement the inverse power method by recognizing that Equation (5.144) is equivalent to

$$\mathbf{A} \left[q^{(m+1)} \mathbf{x}^{(m+1)} \right] = \mathbf{x}^{(m)}. \quad (5.147)$$

This is of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad (5.148)$$

which we investigated earlier with regard to sets of linear equations. Knowing \mathbf{A} and $\mathbf{x}^{(m)}$, Equation (5.147) can be solved for the unknown $q^{(m+1)}\mathbf{x}^{(m+1)}$, by Gaussian elimination, for example. $q^{(m+1)}$ and $\mathbf{x}^{(m+1)}$ are then found by requiring the vector to be normalized. This solution can then be used in the next iteration and the process repeated.

EXERCISE 5.28

Use the inverse power method with Gaussian elimination to solve for the eigenvalue of the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \quad (5.149)$$

with the smallest magnitude. Note that the inverse is never explicitly needed nor calculated.

The power method, or the inverse power method, can be very slow to converge and by itself is usually not particularly useful in finding eigenvalues. However, the method provides an extremely useful technique for finding *eigenvectors* if we already have a good approximation for the eigenvalue. Consider the eigenvalue equation

$$\mathbf{A}\mathbf{u}_j = \lambda_j \mathbf{u}_j. \quad (5.150)$$

If q is an approximation to λ_j , then we can subtract qu_j from both sides of the equation to yield

$$(\mathbf{A} - q\mathbf{I})\mathbf{u}_j = (\lambda_j - q)\mathbf{u}_j, \quad (5.151)$$

where \mathbf{I} is the identity matrix. From this equation we learn that the eigenvalue of $(\mathbf{A} - q\mathbf{I})$ is $(\lambda_j - q)$, and the eigenvector is \mathbf{u}_j . But if q is a good approximation to λ_j , then $(\lambda_j - q)$ is nearly zero! If we now use the inverse power method, the eigenvector should converge *very rapidly* since the dominant eigenvalue of $(\mathbf{A} - q\mathbf{I})^{-1}$ is very large — only a few iterations are needed to obtain a good eigenvector!

With the appropriate q , any eigenvalue λ_j can be shifted into the dominant position of $(\mathbf{A} - q\mathbf{I})$, so that this method can be used to find any eigenvector. If it should happen that \mathbf{A} is tridiagonal, then the solution is further simplified and the subroutine TRISOLVE can be used to find the eigenvector.

EXERCISE 5.29

Consider the nonuniform piano wire again. Impose a grid on the problem, say, every 0.1 meter along its 1-meter length. Since the endpoints are fixed, that leaves nine points along the wire. Solve for the lowest three frequencies, by searching for roots in the determinant. Then determine the eigenvectors associated with them.

EXERCISE 5.30

Whenever a grid is imposed upon a problem, the error associated with its imposition should be investigated — if you don't know how accurate your approximations are, what confidence can you have in your results? Investigate the discretisation error by considering grids with spacing 0.05 meters and 0.025 meters, and compare to the result of the previous exercise. (Both the eigenvalue and eigenvector will differ.)

Finite Elements

In the finite difference approach, a few points were chosen at which the derivatives appearing in the differential equation were approximated by finite differences, and the resulting linear equations solved. The *finite element* method takes a different approach, approximating the exact solution by a (flexible) trial solution, and solving the resulting equations to find the “best” trial function. Both methods have their advantages; certainly, the finite difference approach is easy to understand and is straightforward to program. The finite

element method, on the other hand, is a bit more difficult to grasp and to program, but frequently yields superior results for the same computational effort. In many areas of engineering, such as structural analysis, it has essentially replaced the finite difference method. Its use in physics is not presently widespread, but that circumstance is rapidly changing.

Let's begin by considering a simple differential equation, say,

$$y'' - 6x = 0, \quad (5.152)$$

subject to the boundary conditions

$$y(0) = 0, \quad y(1) = 1. \quad (5.153)$$

(Clearly, the solution to this equation is $y = x^3$.) Let's guess a solution of the form

$$p(x) = \alpha x^2 + \beta x + \gamma. \quad (5.154)$$

Requiring $p(0) = 0$ and $p(1) = 1$, we find that

$$p(x) = \alpha x^2 + (1 - \alpha)x. \quad (5.155)$$

We now have a trial function, parameterized by α , that satisfies the boundary conditions of the problem. Of course, the true solution also satisfies the differential equation $y'' - 6x = 0$. If we were to substitute our trial solution $p(x)$ for $y(x)$, the left side of the differential equation would not be zero — unless we had happened to have guessed the exact solution! — but would be some function

$$R = p'' - 6x = 2\alpha - 6x, \quad (5.156)$$

which we call the *residual error*. In the finite element method the parameters of the residual error — α in this case — are chosen so that this residual error is minimized (in some sense).

For example, we can consider the integral of the square of the error,

$$I = \int_0^1 R^2 dx \quad (5.157)$$

and minimize it with respect to α :

$$\frac{\partial I}{\partial \alpha} = 2 \int_0^1 R \frac{\partial R}{\partial \alpha} dx = 2 \int_0^1 (2\alpha - 6x)(2) dx = 0. \quad (5.158)$$

Evaluating the integral, we find

$$\alpha = \frac{3}{2}, \quad (5.159)$$

so that

$$p(x) = \frac{3x^2 - x}{2} \quad (5.160)$$

is the best quadratic approximation to the solution of the differential equation, in a least squares sense. This approximation is compared to the exact solution in Figure 5.12.

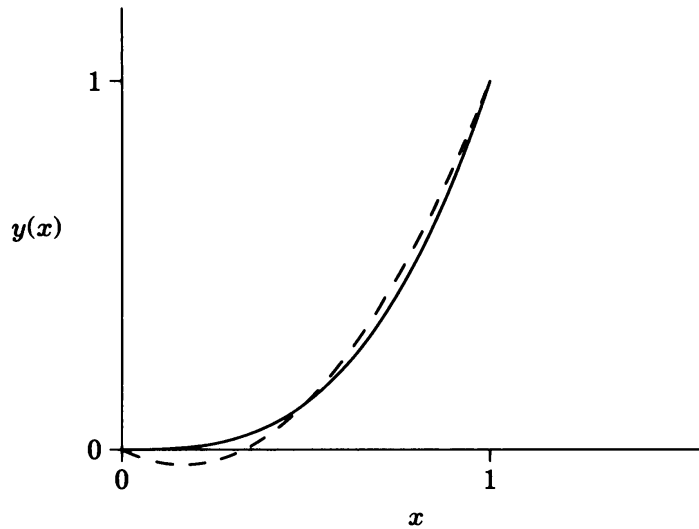


FIGURE 5.12 Comparison of the exact solution of $y'' - 6x = 0$ to a least squares finite element approximation.

The basic advantage that the finite element method enjoys over the finite difference method is thus similar to the advantage that the least squares method has over Lagrange interpolation — rather than being overly concerned with point-by-point agreement with the exact solution, the finite element method achieves good overall agreement by forcing the integral of the error to be small.

To find a better solution to the differential equation we could use a higher order polynomial for the trial function. Alternatively, we could use relatively low order trial functions, and break the region of interest into smaller intervals. Both of these approaches are used, but as a practical matter we would rarely want to use higher than a third-order trial function due to the spurious oscillations that occur with higher order polynomials. Thus a pri-

many avenue to greater accuracy is through using a larger number of intervals.

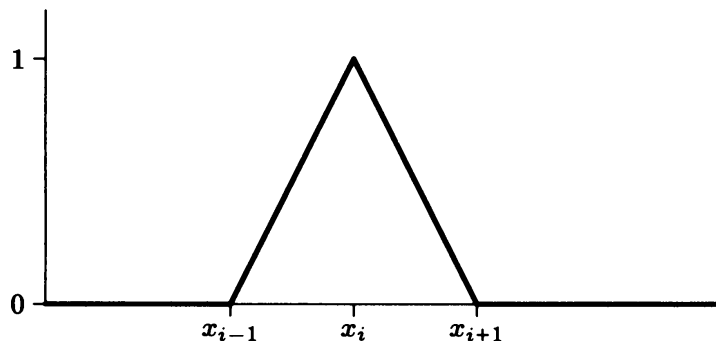


FIGURE 5.13 Plot of the basis function $\phi_i(x)$.

Let's consider an approximation that involves many intervals and is linear in each one, a *piecewise linear* trial function. A particularly elegant way to express this approximation is in terms of the *basis functions* ϕ_i , where

$$\phi_i(x) = \begin{cases} 0, & x \leq x_{i-1} \\ \frac{x - x_{i-1}}{x_i - x_{i-1}}, & x_{i-1} \leq x \leq x_i \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & x_i \leq x \leq x_{i+1} \\ 0, & x_{i+1} \leq x. \end{cases} \quad (5.161)$$

Such a basis function is illustrated in Figure 5.13. (Does this remind you of linear interpolation?) An arbitrary piecewise linear trial function can then be written as

$$p(x) = \sum_{j=0}^N \alpha_j \phi_j(x), \quad (5.162)$$

where the total region has been broken into N intervals. The requirement that

$$p(0) = y(0) \quad \text{and} \quad p(1) = y(1) \quad (5.163)$$

establishes the parameters $\alpha_0 = y(0)$, $\alpha_N = y(1)$, but we're left with $N - 1$ parameters to be determined. These could be determined by the method of least squares, but there are other possibilities. For example, we can require that the average error, weighted by some function w_i , be zero. That is, we can require that

$$\int R(x) w_i(x) dx = 0, \quad i = 1, \dots, N - 1. \quad (5.164)$$

This is the general approach of *weighted residuals*, and is the basis for several versions of the finite element method. Choosing

$$w_i(x) = \frac{\partial R}{\partial \alpha_i} \quad (5.165)$$

we recover the least squares method.

A particularly useful choice of weighting functions are the basis functions $\phi_i(x)$, which then leads to the so-called *Galerkin method*,

$$\int R(x) \phi_i(x) dx = 0, \quad i = 1, \dots, N-1. \quad (5.166)$$

For our example problem, we have

$$\int_0^1 \left(\frac{d^2 p(x)}{dx^2} - 6x \right) \phi_i(x) dx = 0, \quad i = 1, \dots, N-1. \quad (5.167)$$

Recall that $p(x)$ is piecewise linear, composed of a series of straight line segments. On each interval the slope is constant — and hence the second derivative is zero — but the slope changes from one interval to the next so that the second derivative is infinite at the endpoints of an interval. This would appear to be a problem, but it's only an illusion — the derivative *is* infinite, but we're only concerned with its *integral*. Integrating the offending term by parts, we have

$$\int_0^1 \frac{d^2 p}{dx^2} \phi_i(x) dx = \left. \frac{dp}{dx} \phi_i \right|_0^1 - \int_0^1 \frac{dp}{dx} \frac{d\phi_i}{dx} dx, \quad i = 1, \dots, N-1, \quad (5.168)$$

so that we only need to evaluate finite quantities. And with our choice of basis, the first term is zero since $\phi_i(0) = \phi_i(1) = 0$ for $i = 1, \dots, N-1$. The remaining integral is

$$- \int_0^1 \frac{dp}{dx} \frac{d\phi_i}{dx} dx = - \sum_{j=0}^N \alpha_j \int_0^1 \frac{d\phi_j(x)}{dx} \frac{d\phi_i(x)}{dx} dx, \quad i = 1, \dots, N-1. \quad (5.169)$$

One of the real advantages of using a standard basis set such as $\phi_j(x)$ is that we don't need to evaluate these integrals for every new problem. Rather, we can evaluate them *once*, and use those evaluations in subsequent problems. If the x_i are uniformly spaced between the limits a and b , with $h = x_i - x_{i-1}$,

then

$$\int_a^b \frac{d\phi_i(x)}{dx} \frac{d\phi_j(x)}{dx} dx = \begin{cases} \frac{2}{h}, & |i-j| = 0, \\ -\frac{1}{h}, & |i-j| = 1, \\ 0, & |i-j| > 1. \end{cases} \quad (5.170)$$

Thus

$$\int_0^1 \frac{dp(x)}{dx} \frac{d\phi_i}{dx} dx = \sum_{j=0}^N \alpha_j \int_0^1 \frac{d\phi_j(x)}{dx} \frac{d\phi_i(x)}{dx} dx = \frac{-\alpha_{i-1} + 2\alpha_i - \alpha_{i+1}}{h}, \quad (5.171)$$

and Equation (5.167) becomes

$$\frac{\alpha_{i-1} - 2\alpha_i + \alpha_{i+1}}{h} - \int_0^1 6x\phi_i(x) dx = 0, \quad i = 1, \dots, N-1. \quad (5.172)$$

In this example, the remaining integral can be performed analytically,

$$\int_0^1 x\phi_i(x) dx = hx_i, \quad i = 1, \dots, N-1. \quad (5.173)$$

For other differential equations, or for a different choice of basis functions, it may be necessary to integrate such a term numerically. For the smoothly varying functions that commonly appear in these applications, Gauss–Legendre quadrature is often the method of choice.

With our evaluation of the integrals, Equation (5.172) thus becomes

$$\frac{\alpha_{i-1} - 2\alpha_i + \alpha_{i+1}}{h} = 6hx_i, \quad i = 1, \dots, N-1. \quad (5.174)$$

This we recognize as one component of the matrix equation

$$\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{N-2} \\ \alpha_{N-1} \end{bmatrix} = \begin{bmatrix} 6h^2x_1 - \alpha_0 \\ 6h^2x_2 \\ \vdots \\ 6h^2x_{N-2} \\ 6h^2x_{N-1} - \alpha_N \end{bmatrix}. \quad (5.175)$$

This tridiagonal system can now be solved with the familiar subroutine TRI-SOLVE for the unknown α_i .

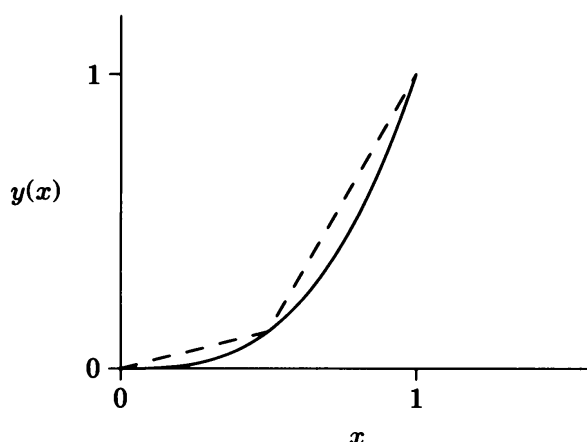


FIGURE 5.14 Comparison of the exact solution of $y'' - 6x = 0$ to a two-interval piecewise linear approximation obtained with the Galerkin finite element method.

Choosing $N = 2$ leads to the single equation $\alpha_1 = 0.125$, yielding the piecewise linear approximation appearing in Figure 5.14. As we would expect, this result is not as accurate as the quadratic least squares approximation. However, it's relatively easy to improve the result by using more intervals in the piecewise approximation.

EXERCISE 5.31

Use a piecewise linear approximation over 10 intervals to approximate better the solution to the differential equation. Plot the result, comparing the approximation to the exact solution.

An Eigenvalue Problem

The finite element method can also be applied to eigenvalue problems. While the derivation of the equations to be solved differs substantially from the derivation with finite differences, the resulting equations are quite similar in appearance. This is unfortunate, in that the principles underlying the two approaches are really quite different.

Let's investigate the vibrating string problem, governed by the differential equation

$$\frac{d^2 y}{dx^2} + \frac{\omega^2 \mu(x)}{T} y = 0, \quad (5.176)$$

and approximate $y(x)$ by the piecewise linear approximation

$$p(x) = \sum_{j=0}^N \alpha_j \phi_j(x). \quad (5.177)$$

α_0 and α_N are again set by the boundary conditions, $y(0) = y(L) = 0$, so that $\alpha_0 = \alpha_N = 0$. The remaining $N - 1$ coefficients α_i are determined from the Galerkin condition,

$$\int_0^L R(x) \phi_i(x) dx = 0, \quad i = 1, \dots, N - 1, \quad (5.178)$$

where the residual error is

$$R(x) = \frac{d^2 p}{dx^2} + \frac{\omega^2 \mu(x)}{T} p. \quad (5.179)$$

Thus the α_i satisfy

$$\int_0^L \left[\frac{d^2 p}{dx^2} + \frac{\omega^2 \mu}{T} p \right] \phi_i dx = 0, \quad i = 1, \dots, N - 1. \quad (5.180)$$

We'll again use integration by parts to evaluate the integral of the second derivative,

$$\int_0^L \frac{d^2 p}{dx^2} \phi_i(x) dx = \left. \frac{dp}{dx} \phi_i \right|_0^L - \int_0^L \frac{dp}{dx} \frac{d\phi_i}{dx} dx, \quad i = 1, \dots, N - 1. \quad (5.181)$$

Due to the choice of basis functions, the first term is zero. We now express $p(x)$ in terms of $\phi_j(x)$ and find that the Galerkin condition becomes

$$-\sum_{j=0}^N \alpha_j \int_0^L \frac{d\phi_j}{dx} \frac{d\phi_i}{dx} + \sum_{j=0}^N \alpha_j \frac{\omega^2}{T} \int_0^L \mu(x) \phi_j(x) \phi_i(x) dx = 0, \quad i = 1, \dots, N - 1. \quad (5.182)$$

The first integral we've already done, with the result that

$$-\sum_{j=0}^N \alpha_j \int_0^L \frac{d\phi_j}{dx} \frac{d\phi_i}{dx} = \frac{-\alpha_{i-1} + 2\alpha_i - \alpha_{i+1}}{h}. \quad (5.183)$$

Let's approximate $\mu(x)$ by its value at x_i , μ_i . Then

$$\int_0^L \mu(x) \phi_j(x) \phi_i(x) dx \approx \mu_i \int_0^L \phi_j(x) \phi_i(x) dx, \quad i = 1, \dots, N - 1. \quad (5.184)$$

We again note that the necessary integrals involving the basis functions $\phi_i(x)$ can be performed once, and subsequently used whenever needed. Using the explicit expression for the basis functions, we find

$$\int_0^L \phi_j(x) \phi_i(x) dx = \begin{cases} 0, & |i - j| > 1, \\ \frac{h}{6}, & |i - j| = 1, \\ \frac{2h}{3}, & |i - j| = 0. \end{cases} \quad (5.185)$$

We could do a better job of approximating this integral, if we so desire. To illustrate, we'll leave $\mu(x)$ arbitrary and evaluate the integral by Gaussian quadrature. Recall that an n -point quadrature is exact for polynomials up to order $(2n - 1)$. Since the basis functions are linear, a two-point Gaussian quadrature will yield exact results for a linear μ . There are three cases, $j = i - 1, i, i + 1$. For the first case, the product $\phi_{i-1}\phi_i$ is zero everywhere except on the interval between x_{i-1} and x_i , so that

$$\begin{aligned} \int_0^L \mu(x) \phi_{i-1}(x) \phi_i(x) dx &= \int_{x_{i-1}}^{x_i} \mu(x) \phi_{i-1}(x) \phi_i(x) dx \\ &= \int_{x_{i-1}}^{x_i} \mu(x) \frac{x_i - x}{h} \frac{x - x_{i-1}}{h} dx \\ &= \frac{h}{2} \int_{-1}^1 \mu\left(\frac{hy}{2} + x_{i-1} + \frac{h}{2}\right) \frac{1}{2}(1 - y) \frac{1}{2}(y - 1) dy, \end{aligned} \quad (5.186)$$

where we've made the substitution

$$y = [x - (x_{i-1} + h)] \frac{2}{h}. \quad (5.187)$$

The integral is now in the form required for Gauss–Legendre integration. Using only a two-point quadrature, we have

$$\begin{aligned} \int_0^L \mu(x) \phi_{i-1}(x) \phi_i(x) dx &\approx \frac{h}{12} \left[(-2 - \sqrt{3}) \mu\left(x_{i-1} + \frac{h}{2} - \frac{h}{2\sqrt{3}}\right) \right. \\ &\quad \left. + (-2 + \sqrt{3}) \mu\left(x_{i-1} + \frac{h}{2} + \frac{h}{2\sqrt{3}}\right) \right]. \end{aligned} \quad (5.188)$$

Clearly, the case $j = i + 1$ will proceed in a nearly identical manner. For the case $j = i$, both the intervals $x_{i-1} \leq x \leq x_i$ and $x_i \leq x \leq x_{i+1}$ need to be considered. Since the basis functions change radically from one interval to

the next, while Gauss–Legendre integration presumes a smoothly varying integrand, the two intervals should be considered separately. But we see that performing these integrals, either analytically or numerically, is easily accomplished.

Let's return to our simplified evaluation of the integrals, as given in Equation (5.185). The Galerkin conditions of Equation (5.182) then read

$$\frac{-\alpha_{i-1} + 2\alpha_i - \alpha_{i+1}}{h} + \frac{\omega^2 \mu_i}{T} \left[\alpha_{i-1} \frac{h}{6} + \alpha_i \frac{2h}{3} + \alpha_{i+1} \frac{h}{6} \right] = 0, \quad i = 1, 2, \dots, N-1. \quad (5.189)$$

The comparison of this equation with its finite difference counterpart, Equation (5.116), is rather interesting. The equations appear rather similar — certainly, the derivatives appear in the same fashion. But note that the term in the brackets represents a weighted average of the α 's, with exactly the same coefficients as obtained in Simpson's rule integration! Where the finite difference method simply has a function evaluation, the finite element method has the *average* of the function, as evaluated by integrating over the neighboring intervals!

We can write these equations in matrix form as

$$\begin{bmatrix} -2\frac{T}{\mu_1 h^2} & \frac{T}{\mu_1 h^2} & 0 & & \\ \frac{T}{\mu_1 h^2} & -2\frac{T}{\mu_1 h^2} & \frac{T}{\mu_1 h^2} & & \\ & & \ddots & \ddots & \\ & & & \frac{T}{\mu_1 h^2} & -2\frac{T}{\mu_1 h^2} & \frac{T}{\mu_1 h^2} \\ & & & \frac{T}{\mu_1 h^2} & -2\frac{T}{\mu_1 h^2} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{N-2} \\ \alpha_{N-1} \end{bmatrix} = \frac{\omega^2}{6} \begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{N-2} \\ \alpha_{N-1} \end{bmatrix}. \quad (5.190)$$

In matrix notation, we write this as

$$\mathbf{Ax} = \lambda \mathbf{Bx}, \quad (5.191)$$

the *generalized* eigenvalue problem. If we're seeking the smallest eigenvalue, then a modified version of the inverse iteration method can be used. And since the problem is tridiagonal, we can again use TRISOLVE in its solution.

We begin with an initial $\mathbf{x}^{(0)}$, and evaluate the product $\mathbf{B}\mathbf{x}^{(0)}$. Then TRISOLVE is used to solve the equation

$$\mathbf{A}\mathbf{x}^{(1)} = (\mathbf{B}\mathbf{x}^{(0)}) \quad (5.192)$$

for the unknown (and unnormalized) $\mathbf{x}^{(1)}$. In general, the iteration proceeds according to

$$\mathbf{A}\mathbf{x}^{(k)} = \mathbf{B}\mathbf{x}^{(k-1)}. \quad (5.193)$$

Note that the normalization of $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k-1)}$ is not the same.

The generalized eigenvalue equation itself provides an exact expression for the eigenvalue in terms of the eigenvector. Multiplying Equation (5.191) by \mathbf{x}^T , we have

$$\mathbf{x}^T \mathbf{A}\mathbf{x} = \lambda \mathbf{x}^T \mathbf{B}\mathbf{x}, \quad (5.194)$$

or

$$\lambda = \frac{\mathbf{x}^T \mathbf{A}\mathbf{x}}{\mathbf{x}^T \mathbf{B}\mathbf{x}}. \quad (5.195)$$

With this hint, we define the *Rayleigh quotient* as

$$R = \frac{\mathbf{x}^{(k)T} \mathbf{A}\mathbf{x}^{(k)}}{\mathbf{x}^{(k)T} \mathbf{B}\mathbf{x}^{(k)}}. \quad (5.196)$$

As $k \rightarrow \infty$, $R \rightarrow \lambda$. In practice, the iteration is terminated when two successive iterations agree to within some predetermined tolerance. Of course, the vector needs to be renormalized after every iteration. However, rather than requiring $\mathbf{x}^T \mathbf{x} = 1$, in this problem we require a different type of normalization,

$$\mathbf{x}^T \mathbf{B}\mathbf{x} = 1. \quad (5.197)$$

An appropriate program then looks like

```
double precision a(11),b(11),c(11),lambda
double precision bb(11), X0(11),X1(11)
double precision h,ans,x,sum,T,mu, quotient, up, down
integer i, count, N
logical done
parameter( n = 9, h = 0.1d0, T = 300.d0, mu=0.01d0 )

done = .FALSE.
```

```

      DO i = 1, N
        a(i) = -1.d0*t/(mu*h*h)
        b(i) = 2.d0*t/(mu*h*h)
        c(i) = -1.d0*t/(mu*h*h)
      END DO

*
* X0 is x(k-1) --- initially, elements are just 1/sqrt(N)
*
      DO i = 1, n
        X0(i) = 1.d0/SQRT(DBLE(N))
      END DO

      count = 0

*
* Initializing for the first pass --- let X1 be A x(0).
* (This is ONLY for the first pass!!!)
*
      X1(1) = b(1) * X0(1) + c(1) * X0(2)
      DO i = 2, N-1
        X1(i) = a(i)*X0(i-1) + b(i)*X0(i) + c(i)*X0(i+1)
      END DO
      X1(N) = a(N) * X0(N-1) + b(N) * X0(N)

*
* This is the top of the iteration loop.  <----- LOOK HERE
*
*
* Evaluate x(k-1) * [ A x(k-1) ] --- result in UP
*
1000  up = 0.d0
      DO i = 1, N
        up = up + X0(i) * X1(i)
      END DO

*
* Evaluate the product [ B x(k-1) ], store in X1
*
      X1(1) = (4.d0*X0(1)+ X0(2) )/6.d0
      DO i = 2, N-1
        X1(i) = (X0(i-1)+4.d0*X0(i)+X0(i+1))/6.d0
      END DO
      X1(N) = (X0(N-1)+4.d0*X0(N))/6.d0

*
* Evaluate x(k-1) B x(k-1), the denominator
*
      down = 0.d0

```

```

      DO i = 1, N
        down = down + X0(i)*X1(i)
      END DO
*
* Renormalize the vector in X1
*
      DO i = 1, N
        X1(i) = X1(i) / sqrt(down)
      END DO
*
* Compute the Rayleigh quotient, UP/DOWN, and iterate until
* accurate result is obtained.
*
      IF (count .ne. 0) THEN
        error = (up/down - quotient) / (up/down)
        if ( abs(error) .le. tolerance) done = .TRUE.
      ENDIF
      quotient = up/down
      count = count + 1
      write(*,*)'approximate eigenvalue =',ratio
*
* Solve the tridiagonal system for X0, the next
* approximate eigenvector.
*
      CALL TRISOLVE( A, B, C, X0, X1, N)

      if(.not.done) goto 1000
      omega = sqrt ( quotient ) ! calculate the frequency

end

```

EXERCISE 5.32

Verify that the finite element program we've presented does in fact solve the vibrating string problem. Then modify the program to consider 20 intervals in the piecewise linear approximation, rather than 10, and compare your results.

EXERCISE 5.33

Consider the second string problem, with $\mu(x) = \mu_o + (x - L/2)\Delta$. You will need to evaluate the integrals involving $\mu(x)$ — you should be able to do these analytically. What's the fundamental frequency of

the vibration? How does this result compare — at the same N — to the finite difference result?

EXERCISE 5.34

Integrals should be performed analytically whenever possible. However, situations arise in which numerical evaluations are justified. Replace the analytic evaluation of integrals in your program by two-point Gauss–Legendre integration. Verify that your replacement is correct by comparing results with the previous exercise. Then, consider a different density function,

$$\mu(x) = \mu_0 + \Delta(x - L/2)^2. \quad (5.198)$$

What is the fundamental frequency of vibration? What is the shape of the string, compared to the two other strings we've been investigating?

References

Many of the topics discussed in this chapter are routinely covered in numerical analysis texts. There are many excellent ones to choose from; the following is just a brief list of the ones I particularly like.

Forman S. Acton, *Numerical Methods That Work*, Harper & Row, New York, 1970. This book is particularly commendable with regard to Acton's philosophy toward computation.

Anthony Ralston, *A First Course in Numerical Analysis*, McGraw–Hill, New York, 1965.

Richard L. Burden and J. Douglas Faires, *Numerical Analysis*, Prindle, Weber & Schmidt, Boston, 1985.

Curtis F. Gerald and Patrick O. Wheatley, *Applied Numerical Analysis*, Addison–Wesley, Reading, Massachusetts, 1989.

J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer–Verlag, New York, 1980. This text is not as introductory as the title suggests, but it is an excellent reference.

For those interested in the music of stringed instruments, I highly recommend

The Journal of the Catgut Acoustical Society. There are, of course, several texts on “the physics of music,” including

Thomas D. Rossing, *The Science of Sound*, Addison–Wesley, Reading, Massachusetts, 1990.

Arthur H. Benade, *Fundamentals of Musical Acoustics*, Oxford University Press, New York, 1976.

Chapter 6:

Fourier Analysis

By now, we have all come to appreciate the power and usefulness of the Taylor series, which allows us to expand (most) continuous functions in an infinite series of terms. While Taylor series expressions are very useful in general, there are two situations for which they are not well suited. The first is for periodic functions — the Taylor series utilizes a unique point, about which the series is expanded. But in a periodic function, equivalent points lie just one period away. This inherent property of a periodic function is simply not taken into account in the Taylor series. The second situation that causes problems is in describing a function with discontinuities — Taylor’s series requires the function, and *all* of its derivatives, to exist. These requirements result from the derivation of Taylor’s series from the Laurent series, which in turn involves the analyticity of the function in the complex plane. Since analyticity is a global property, it’s not surprising that the derived Taylor series requires all those derivatives. So the Taylor series can never be used describe a function with jumps, or even with jumps in some derivative.

In modern terms, the difficulty actually lies in the concept of a *function* itself. Euler had an elementary view, that a function should be representable by a smoothly drawn curve. This is in significant contrast to the modern concept, which simply associates each member of a set $\{x\}$ with an element of the set $\{f(x)\}$. And bridging the gap lies the work of Fourier, Cauchy, and Riemann. While Cauchy and Riemann were primarily mathematicians, Fourier belonged to that element of the French school that believed mathematics should be applied to real world problems — indeed, modern methods of analysis are deeply dependent upon the work of Fourier, including a sizable chunk of physics.

The Fourier Series

Both Euler and d’Alembert had solved the “vibrating string” problem using

two arbitrary functions, while Daniel Bernoulli had found a solution in terms of an infinite series of trigonometric functions. Since the trigonometric functions seemed to imply a periodic solution, Bernoulli's solution appeared less general. Fourier's contribution was the idea that any function can be expressed in the form

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos nt + \sum_{n=1}^{\infty} b_n \sin nt. \quad (6.1)$$

Clearly, this expansion can describe periodic functions. And in contrast to the Taylor series, a Fourier series can be used to describe a discontinuous function, or a function with discontinuous derivatives. Jean Baptiste Joseph Fourier (1768–1830), a teacher of mathematics at the École Normale and the École Polytechnique in Paris and an administrator under Napoleon Bonaparte, submitted an essay on the mathematical theory of heat to the French Academy of Science in 1812. Although he won the Academy prize, the essay was criticized for a certain looseness of reasoning by the panel of referees, Lagrange, Laplace, and Legendre. Lagrange, in particular, did not believe that the series converged! It was not until 1828 that Dirichlet found the mathematically *sufficient* conditions that can be imposed upon the function to insure convergence, but it is not known if they are *necessary*. And even then, the series may not converge to the value of the function from which it is derived!

Dirichlet's theorem: *If $f(t)$ is periodic of period 2π , if for $-\pi < t < \pi$ the function $f(t)$ has a finite number of maximum and minimum values and a finite number of discontinuities, and if $\int_{-\pi}^{\pi} f(t) dt$ is finite, then the Fourier series converges to $f(t)$ at all points where $f(t)$ is continuous, and at jump-points it converges to the arithmetic mean of the right-hand and left-hand limits of the function.*

In passing, we note that Bernhard Riemann tried to make Dirichlet's conditions less constraining, only to be led down the path leading toward a new definition of the integral: the Riemann integral of conventional calculus. Along the way, he showed that a function $f(x)$ may be integrable and not have a Fourier series representation. The famous Russian mathematician and physicist Andrey Kolmogorov constructed an integrable function with a Fourier series divergent almost everywhere — at the advanced age of 19.

For our purposes, we'll assume that the Fourier series converges for any problem of interest to us; any that are not convergent should promptly be marked *pathological* and sent to the nearest Department of Applied Mathematics.

At a fundamental level, the Fourier series works because the sines and cosines form a complete set, so that they may be used to describe *any* function, and because they are orthogonal on any 2π interval. Equation (6.1) is actually a statement of completeness; the orthogonality is expressed by

$$\int_{-\pi}^{\pi} \sin mt \sin nt \, dt = \begin{cases} \pi \delta_{m,n}, & m \neq 0, \\ 0, & m = 0, \end{cases} \quad (6.2)$$

$$\int_{-\pi}^{\pi} \cos mt \cos nt \, dt = \begin{cases} \pi \delta_{m,n}, & m \neq 0, \\ 2\pi, & m = n = 0, \end{cases} \quad (6.3)$$

$$\int_{-\pi}^{\pi} \sin mt \cos nt \, dt = 0, \quad \text{all integral } m \text{ and } n. \quad (6.4)$$

Using these orthogonality relations, we can easily find that the coefficients are derived from the function by

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos nt \, dt, \quad (6.5)$$

and

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin nt \, dt. \quad (6.6)$$

As an example, consider the unit step function

$$f(t) = \begin{cases} -1, & t < 0 \\ +1, & t > 0. \end{cases} \quad (6.7)$$

Since the function is odd, all the a_n must be zero. The b_n are easily found to be

$$\begin{aligned} b_n &= \frac{1}{\pi} \int_{-\pi}^0 (-1) \sin nt \, dt + \frac{1}{\pi} \int_0^{\pi} (+1) \sin nt \, dt \\ &= \frac{2}{\pi} \int_0^{\pi} \sin nt \, dt \\ &= \frac{2}{n\pi} [1 - \cos n\pi] \\ &= \begin{cases} 0, & n = 2, 4, 6, \dots \\ 4/n\pi, & n = 1, 3, 5, \dots \end{cases} \end{aligned} \quad (6.8)$$

In practice, only a finite number of terms are included in the Fourier series expansion of a function. Since the sines and cosines form an orthogonal set, our

remarks made in Chapter 3 concerning approximation by orthogonal functions apply. In particular, we know that these coefficients provide the best possible fit to the original function, in a least squares sense. As more terms are retained in the series, the fit becomes even better. A sequence of such approximations is shown in Figure 6.1.

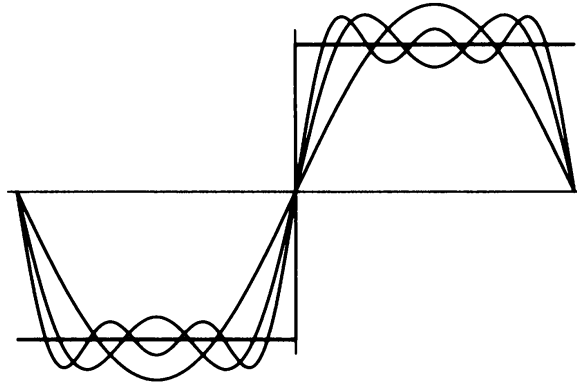


FIGURE 6.1 Convergence of the Fourier series representation of a unit step function, retaining 1, 2, and 3 terms in the series.

As the number of terms increases, the approximation does a better and better job of describing the function. Where the function is continuous, the approximations oscillate about the true values of the function, and as more terms are retained the oscillations decrease in magnitude. At the discontinuity, the approximation gives the mean value of the left and right limits of the function, just as Dirichlet said it would. But near the discontinuity, the oscillations are not decreasing as rapidly as elsewhere. This is called the *overshoot*, and in some sense is the price we pay for being able to describe the discontinuity at all. (Remember, Taylor's series is virtually useless in such situations.) One might think that as more terms are retained in the series, the overshoot would disappear, but that's not the case.

EXERCISE 6.1

Examine the overshoot as the number of terms in the series increases. Evaluate the approximation, and hence the overshoot, in the vicinity of the discontinuity $0 \leq t \leq 0.1$, retaining 10, 20, 30, 40, and 50 terms in the series. The persistence of the overshoot is known as the Gibbs phenomenon.

As another example, consider the function

$$f(t) = |t|, \quad -\pi \leq t \leq \pi. \quad (6.9)$$

Because this function is even, its Fourier series representation will contain only cosine terms, e.g., all the b_n are zero.

EXERCISE 6.2

Evaluate the coefficients of the Fourier series of $f(t) = |t|$. Investigate the convergence of the series by plotting the approximation on the interval $-\pi \leq t \leq \pi$ using N terms, with $N = 2, 4, 6, 8$, and 10 . For comparison, plot the original function as well. How rapidly does this series converge, compared to the series describing the unit step function?

While much of our work will be with real functions, the concept of Fourier series is applicable to complex functions as well. And sometimes, it's simply more convenient to use a complex representation. Expressing the sines and cosines as exponentials, Equation (6.1) can be rewritten as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{int} \quad (6.10)$$

in which

$$c_n = \begin{cases} (a_n - ib_n)/2, & n > 0, \\ a_0/2, & n = 0, \\ (a_{|n|} + ib_{|n|})/2, & n < 0. \end{cases} \quad (6.11)$$

The c_n 's can, of course, be obtained by integration,

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-int} dt. \quad (6.12)$$

The Fourier Transform

Closely related to the idea of the Fourier series representation of a function is the Fourier transform of a function. The series representation is useful in describing functions over a limited region, or on the infinite interval $(-\infty, \infty)$ if the function is periodic. Fourier transforms, on the other hand, are useful in describing nonperiodic functions on the infinite interval.

To develop the transform, let's first consider the series representation of a function that is periodic on the interval $[-T, T]$. Making the substitution

$t \rightarrow \pi t/T$, we have

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\pi t/T}, \quad (6.13)$$

where

$$c_n = \frac{1}{2T} \int_{-T}^T f(t) e^{-in\pi t/T} dt. \quad (6.14)$$

We can now identify the discrete frequencies appearing in the summations as being

$$\omega = \frac{n\pi}{T}, \quad (6.15)$$

and the differences between successive frequencies as being

$$\Delta\omega = \frac{\pi}{T}. \quad (6.16)$$

Then the series can be written as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\Delta\omega t}, \quad (6.17)$$

where

$$c_n = \frac{\Delta\omega}{2\pi} \int_{-T}^T f(t) e^{-in\Delta\omega t} dt. \quad (6.18)$$

We now define

$$c_n = \frac{\Delta\omega}{\sqrt{2\pi}} g(n\Delta\omega) \quad (6.19)$$

so that

$$g(n\Delta\omega) = \frac{1}{\sqrt{2\pi}} \int_{-T}^T f(t) e^{-in\Delta\omega t} dt \quad (6.20)$$

and

$$f(t) = \frac{1}{\sqrt{2\pi}} \sum_{n=-\infty}^{\infty} \Delta\omega g(n\Delta\omega) e^{in\Delta\omega t}. \quad (6.21)$$

We now take the limit as $T \rightarrow \infty$. In so doing, $n\Delta\omega$ becomes the continuous variable ω and the summation in Equation (6.21) becomes an integral. Thus

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{i\omega t} d\omega \quad (6.22)$$

and

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt. \quad (6.23)$$

We now *define* $g(\omega)$ to be the Fourier transform of $f(t)$,

$$\mathcal{F}[f(t)] = g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t)e^{-i\omega t} dt, \quad (6.24)$$

and $f(t)$ to be the *inverse* transform of $g(\omega)$,

$$\mathcal{F}^{-1}[g(\omega)] = f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega)e^{i\omega t} d\omega. \quad (6.25)$$

According to our definitions, the factor of 2π is distributed symmetrically between the transform and its inverse. This is only a convention, and one that (unfortunately) is not followed universally. Furthermore, there's nothing *special* or *distinguishing* about our choice of time as being the “original” variable — ω could just as easily have been chosen, in which case the definitions of transform and inverse transform would be reversed. Whatever convention is chosen, you must of course remain consistent. Considerable caution should be exercised concerning these points in order to avoid confusion and (potentially) a great loss of time and effort.

EXERCISE 6.3

Write a program to evaluate numerically the Fourier transform of the function

$$f(t) = \begin{cases} a(1 - a|t|), & |t| < \frac{1}{a} \\ 0, & |t| > \frac{1}{a}, \end{cases}$$

with $a = 10$, for both positive and negative values of ω . Perform the analytic integrals to check your numerical work.

EXERCISE 6.4

Consider the function of the previous exercise. To help visualize the transform, to see which frequencies are most important, and to see how the transform depends upon the original function, evaluate the transform of the function with $a = 2, 4, 6, 8$, and 10 and plot its magnitude, $|g(\omega)|$.

Properties of the Fourier Transform

There's considerable symmetry in the Fourier transform pair, and they possess several fundamental properties that are significant. From a formal point of view, perhaps the most fundamental property is that the Fourier transform

is linear. That is, if $f_1(t)$ and $f_2(t)$ are two functions having Fourier transforms $g_1(\omega)$ and $g_2(\omega)$, then the Fourier transform of $f_1(t) + f_2(t)$ is

$$\begin{aligned} g(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} [f_1(t) + f_2(t)] e^{-i\omega t} dt \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f_1(t) e^{-i\omega t} dt + \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f_2(t) e^{-i\omega t} dt \\ &= g_1(\omega) + g_2(\omega). \end{aligned} \quad (6.26)$$

Another property is the scaling relation. Let's imagine that $f(t)$ and $g(\omega)$ are Fourier transforms of one another. For α positive,

$$\begin{aligned} \mathcal{F}[f(\alpha t)] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(\alpha t) e^{-i\omega t} dt \\ &= \frac{1}{\sqrt{2\pi}} \frac{1}{\alpha} \int_{-\infty}^{\infty} f(t') e^{-i\omega t'/\alpha} dt' \\ &= \frac{1}{\alpha} g\left(\frac{\omega}{\alpha}\right), \quad \alpha > 0, \end{aligned} \quad (6.27)$$

where the substitution $t' = \alpha t$ was made. But if α is negative, the limits of integration are reversed — the same substitution leads to

$$\begin{aligned} \mathcal{F}[f(\alpha t)] &= \frac{1}{\sqrt{2\pi}} \frac{1}{\alpha} \int_{\infty}^{-\infty} f(t') e^{-i\omega t'/\alpha} dt' \\ &= -\frac{1}{\sqrt{2\pi}} \frac{1}{\alpha} \int_{-\infty}^{\infty} f(t') e^{-i\omega t'/\alpha} dt' \\ &= -\frac{1}{\alpha} g\left(\frac{\omega}{\alpha}\right), \quad \alpha < 0. \end{aligned} \quad (6.28)$$

These results can be combined to yield the general expression

$$\mathcal{F}[f(\alpha t)] = \frac{1}{|\alpha|} g\left(\frac{\omega}{\alpha}\right). \quad (6.29)$$

EXERCISE 6.5

Show that there's a similar relation for inverse transforms,

$$\mathcal{F}^{-1}[g(\beta\omega)] = \frac{1}{|\beta|} f\left(\frac{t}{\beta}\right). \quad (6.30)$$

Equations (6.29) and (6.30) are known as *scaling* relations, and provide a key insight into the Fourier transform pair. In Exercise 6.4, you found that as $f(t)$ became broader, $g(\omega)$ became narrower. The essence of this behavior is contained in these scaling relations. Consider $f(\alpha t)$, such as in Figure 6.2. As α is increased, the function becomes narrower in time. Simultaneously, its Fourier transform becomes broader in frequency!

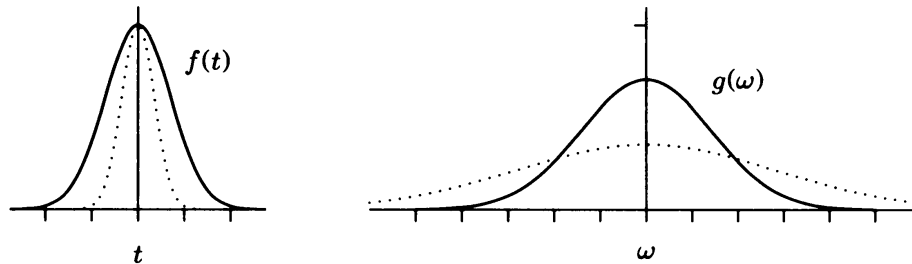


FIGURE 6.2 The function $f(t)$ and its Fourier transform $g(\omega)$ are plotted by the solid lines. As f is scaled to become narrower, as indicated by the dotted curve, g becomes broader.

Let's investigate this further and consider a simple example function: the sine function. With only one frequency component, it extends over all of space. If another sine function of the appropriate frequency is added to the first, the sum can be made to cancel in some regions of space while in other regions they will add. With only a few sine and cosine functions, we're limited in what shapes can be reproduced — as with the unit step function, a few terms will give the correct general shape, but the approximation will have oscillations about the true value of the function. Many terms are needed in order to cancel all the oscillations and give an accurate representation of the function, so that an arbitrary function can be described. That's the essence of Fourier convergence — in large part, the role of the additional functions is to cancel the oscillations. Thus, the more localized a function is in time, the more delocalized it is in frequency.

This is more than a casual observation — it's a fundamental property of Fourier transforms, and has direct physical consequences. Usually stated in terms of position and momentum rather than time and frequency, the statement is that the product of the width of the function, Δx , and the width of the transform of the function, Δp , is always greater than or equal to a specific *nonzero* value, \hbar — Heisenberg's uncertainty principle.

There are also *shifting* relations. For example, the Fourier transform

of $f(t - t_0)$ is simply

$$\begin{aligned}
 \mathcal{F}[f(t - t_0)] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t - t_0) e^{-i\omega t} dt \\
 &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(\tau) e^{-i\omega(\tau+t_0)} d\tau \\
 &= e^{-i\omega t_0} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(\tau) e^{-i\omega \tau} d\tau \\
 &= e^{-i\omega t_0} g(\omega),
 \end{aligned} \tag{6.31}$$

where we introduced the variable $\tau = t - t_0$. Likewise, we have the inverse relation

$$\mathcal{F}^{-1}[g(\omega - \omega_0)] = e^{i\omega_0 t} f(t). \tag{6.32}$$

Time reversal can also be of interest, but this can be viewed as an application of the scaling relation, Equation (6.29), with $\alpha = -1$:

$$\mathcal{F}[f(-t)] = \frac{1}{|-1|} g\left(\frac{\omega}{-1}\right) = g(-\omega). \tag{6.33}$$

Additional properties of the transform pair are associated with particular symmetries of $f(t)$. Consider the relation

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{6.34}$$

and its complex conjugate

$$g^*(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f^*(t) e^{i\omega t} dt. \tag{6.35}$$

If $f(t)$ is *purely real*, then

$$g^*(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i(-\omega)t} dt = g(-\omega). \tag{6.36}$$

That is, the *real part* of the transform is an even function while the *imaginary part* is odd. Conversely, if $f(t)$ is *purely imaginary*, then

$$g^*(\omega) = -\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i(-\omega)t} dt = -g(-\omega), \tag{6.37}$$

or

$$g(-\omega) = -g^*(\omega), \quad (6.38)$$

so that the real part of the transform is odd and the imaginary part is even!

Finally, if $f(t)$ is an even function, then $f(-t) = f(t)$ and from the time reversal property we find that the transform is also even,

$$g(-\omega) = g(\omega). \quad (6.39)$$

Likewise, if $f(t)$ is odd then

$$g(-\omega) = -g(\omega). \quad (6.40)$$

These results can be summarized as follows:

If $f(t)$ is real,	then $\Re g(\omega)$ is even and $\Im g(\omega)$ is odd;
if $f(t)$ is imaginary,	then $\Re g(\omega)$ is odd and $\Im g(\omega)$ is even;
if $f(t)$ is even,	then $g(\omega)$ is even;
if $f(t)$ is odd,	then $g(\omega)$ is odd;
if $f(t)$ is real and even,	then $g(\omega)$ is real and even;
if $f(t)$ is real and odd,	then $g(\omega)$ is imaginary and odd;
if $f(t)$ is imaginary and even,	then $g(\omega)$ is imaginary and even;
if $f(t)$ is imaginary and odd,	then $g(\omega)$ is real and odd.

Since derivatives and differential equations play such a central role in physics, we can anticipate a need to calculate the Fourier transform of a derivative such as

$$\mathcal{F}[f'(t)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f'(t) e^{-i\omega t} dt. \quad (6.41)$$

Integrating by parts, we find

$$\mathcal{F}[f'(t)] = \frac{e^{-i\omega t}}{\sqrt{2\pi}} f(t) \Big|_{-\infty}^{\infty} + \frac{i\omega}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt. \quad (6.42)$$

$f(t)$ must vanish as $t \rightarrow \pm\infty$, else the Fourier transform $g(\omega)$ will not exist; as a consequence, the first term is evaluated as zero, and we find that

$$\mathcal{F}[f'(t)] = i\omega g(\omega), \quad (6.43)$$

so the Fourier transform of a derivative is easy to evaluate. We'll use this very important property later, in the solution of partial differential equations.

There are some interesting integral relations associated with Fourier transforms as well. For example, consider the Fourier transform

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \quad (6.44)$$

and its inverse

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega') e^{i\omega' t} d\omega'. \quad (6.45)$$

Note that the variable of integration has been changed to ω' — since it doesn't appear in the evaluated integral, any symbol can be used. Substituting the second expression into the first, we find

$$\begin{aligned} g(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega') e^{i\omega' t} dt \right] e^{-i\omega t} dt \\ &= \int_{-\infty}^{\infty} \left[\frac{1}{2\pi} \int_{-\infty}^{\infty} e^{i(\omega - \omega') t} dt \right] g(\omega') d\omega', \end{aligned} \quad (6.46)$$

or

$$g(\omega) = \int_{-\infty}^{\infty} \delta(\omega - \omega') g(\omega') d\omega', \quad (6.47)$$

where we've introduced the *Dirac delta function*

$$\delta(\omega - \omega') = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{i(\omega - \omega') t} dt. \quad (6.48)$$

Equation (6.47) is not your typical equation, and you might expect that $\delta(\omega - \omega')$ is not an ordinary function — and you'd be right. To get some idea of what it is, consider the related expression

$$\delta_{\tau}(\omega - \omega') = \frac{1}{2\pi} \int_{-\tau}^{\tau} e^{i(\omega - \omega') t} dt = \frac{\sin(\omega - \omega')\tau}{\pi(\omega - \omega')}. \quad (6.49)$$

This *is* an ordinary function and is plotted in Figure 6.3. In the limit that $\tau \rightarrow \infty$, it would seem that $\delta_{\tau}(\omega - \omega') \rightarrow \delta(\omega - \omega')$. For $\omega - \omega' \approx 0$, we have

$$\begin{aligned} \delta_{\tau}(\omega - \omega') &= \frac{\sin(\omega - \omega')\tau}{\pi(\omega - \omega')} \approx \frac{1}{\pi(\omega - \omega')} \left[\tau(\omega - \omega') - \frac{\tau^3(\omega - \omega')^3}{3!} + \dots \right] \\ &\approx \frac{\tau}{\pi} - \frac{\tau^3}{3!\pi}(\omega - \omega')^2 + \dots, \end{aligned} \quad (6.50)$$

so that $\delta_{\tau}(0) = \tau/\pi$. Thus, as $\tau \rightarrow \infty$, we find the function to be larger and larger in magnitude. Meanwhile, the function drops to zero on either side of

the origin at $\omega - \omega' \approx \pm\pi/\tau$ — judging the width of the function to be the distance between these points, we find it to be $2\pi/\tau$. Thus the magnitude of the function increases linearly with τ while the width is inversely related to τ . The product of height and width, a crude estimate of the area, is a constant!

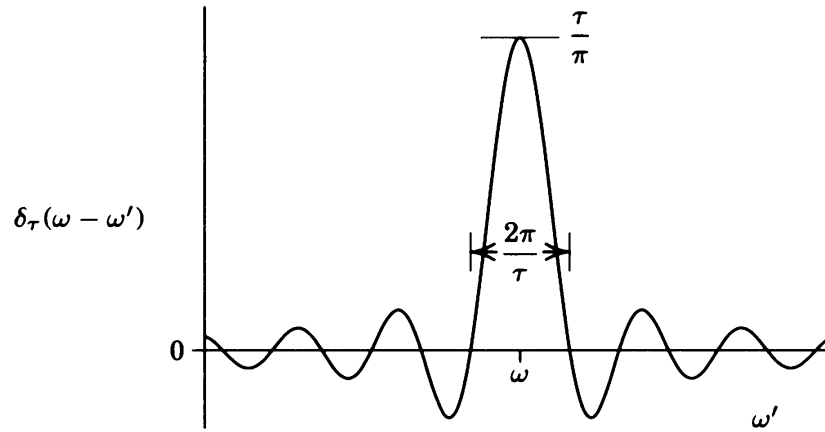


FIGURE 6.3 A plot of $\delta_\tau(\omega - \omega')$. The height of the function is proportional to τ , while the width is inversely proportional to τ . As τ increases, the area under the curve remains (approximately) constant while the curve itself becomes more sharply peaked about $\omega' \approx \omega$.

We can also obtain the integral of the delta function itself. Equation (6.47) is an identity, and so must be true for any function — in particular, it's true for $g(\omega) = 1$, so that

$$\int_{-\infty}^{\infty} \delta(\omega - \omega') d\omega' = 1. \quad (6.51)$$

Thus the integral of the delta function over all ω' is one. And yet, according to Equation (6.48)

$$\delta(0) = \infty. \quad (6.52)$$

This is rather strange behavior for a function. In fact, the delta function is not a function at all in the usual sense. Technically, it's a *distribution*, and only has meaning when it appears in an integrand, as in Equation (6.47). Because of its unique characteristics, the Dirac delta has particular significance in several areas of physics, most notably in quantum mechanics.

Another integral of interest is

$$I = \int_{-\infty}^{\infty} f_1^*(t) f_2(t) dt. \quad (6.53)$$

Writing $f_1(t)$ and $f_2(t)$ in terms of their Fourier transforms, we find

$$\begin{aligned} I &= \int_{-\infty}^{\infty} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g_1(\omega) e^{i\omega t} d\omega \right]^* \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g_2(\omega') e^{i\omega' t} d\omega' \right] dt \\ &= \iint_{-\infty}^{\infty} g_1^*(\omega) g_2(\omega') \left[\frac{1}{2\pi} \int_{-\infty}^{\infty} e^{i(\omega' - \omega)t} dt \right] d\omega d\omega' \\ &= \iint_{-\infty}^{\infty} g_1^*(\omega) g_2(\omega') \delta(\omega - \omega') d\omega d\omega' \\ &= \int_{-\infty}^{\infty} g_1^*(\omega) g_2(\omega) d\omega, \end{aligned} \quad (6.54)$$

which is Parseval's identity. The integrand in these expressions is rather important — in many physical situations, $|f(t)|^2$ is the energy contained in the signal at time t . The quantity $|g(\omega)|^2$ is then the energy contained in the signal in the frequency range between ω and $\omega + d\omega$. For finite duration signals, the function $S(\omega) = |g(\omega)|^2$ is the *energy spectrum*. For periodic (or random) signals, $|f(t)|^2$ is usually a rate of energy flow, or power, and $S(\omega) = |g(\omega)|^2$ is the *power spectrum*. Parseval's identity simply states that the total energy (power) can be obtained by integrating $|f(t)|^2$ over time or $|g(\omega)|^2$ over frequency.

Finally, we need to mention that the Fourier transform can easily be extended to two or more dimensions. For example, in two dimensions we have

$$\mathcal{F}[f(x, y)] = g(k_x, k_y) = \frac{1}{2\pi} \iint_{-\infty}^{\infty} f(x, y) e^{-i(k_x x + k_y y)} dx dy, \quad (6.55)$$

and in three dimensions

$$\begin{aligned} \mathcal{F}[f(x, y, z)] &= g(k_x, k_y, k_z) \\ &= \left(\frac{1}{2\pi} \right)^{\frac{3}{2}} \iiint_{-\infty}^{\infty} f(x, y, z) e^{-i(k_x x + k_y y + k_z z)} dx dy dz. \end{aligned} \quad (6.56)$$

Using vector notation, we recognize this expression as being

$$\mathcal{F}[f(\vec{r})] = g(\vec{k}) = \left(\frac{1}{2\pi} \right)^{\frac{3}{2}} \int f(\vec{r}) e^{-i\vec{k} \cdot \vec{r}} d\vec{r}, \quad (6.57)$$

where the integration extends over all of space. And of course, a function can depend upon time as well as space, so that we might have

$$\mathcal{F}[f(\vec{r}, t)] = g(\vec{k}, \omega) = \left(\frac{1}{2\pi}\right)^2 \int f(\vec{r}, t) e^{-i(\vec{k} \cdot \vec{r} + \omega t)} d\vec{r} dt, \quad (6.58)$$

where the integration extends over all time as well as over all of space.

Convolution and Correlation

Although not immediately obvious, the operations of convolution and correlation are closely associated with Fourier transforms. Their formal definitions are quite similar, although their physical interpretation is quite different. First, let's consider *convolution*.

Consider two functions, $p(t)$ and $q(t)$. Mathematically, the convolution of the two functions is defined as

$$p \otimes q = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p(\tau) q(t - \tau) d\tau. \quad (6.59)$$

Convolutions arise when we try to predict the response of a physical system to a given input. For example, we might have an electronic device that detects and amplifies a voltage signal. Real devices — as opposed to hypothetical ideal ones — always have a certain amount of electrical resistance and capacitance. Let's characterize the device as simply as possible, having only a resistor and a capacitor, as shown in the circuit in Figure 6.4. The signal we *want* to observe is the one input on the left, associated with a “real” physical quantity — the signal we actually see, as a result of our detection, is output on the right as the voltage drop across the capacitor. How is “what we see” related to the “real physical quantity”? Consider what the response of the detector will be. Before the signal arrives, the output of the “detector” is zero; as it arrives, some of its energy goes to charge the capacitor so that the output signal *does not* exactly reproduce the input. Then, after the input signal has passed, the circuit will dissipate its stored energy into the output signal, and so a voltage will persist after the input signal has passed.

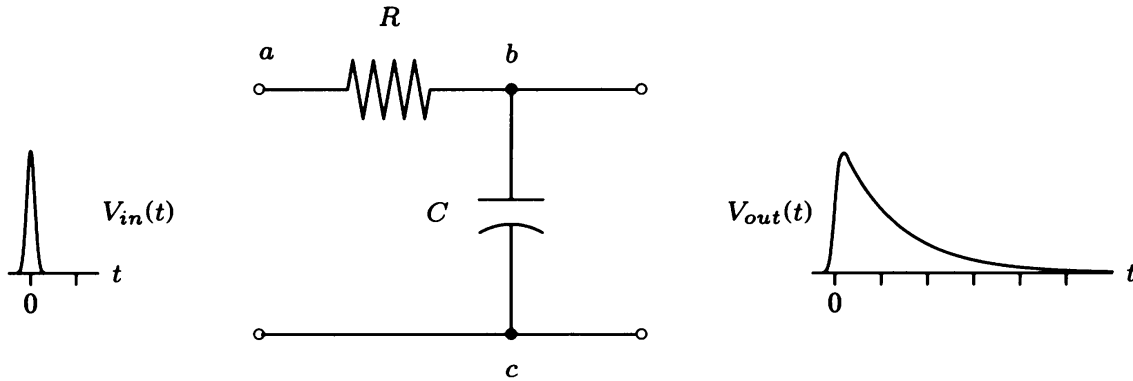


FIGURE 6.4 A simple electrical circuit having resistance R and capacitance C . The response of the circuit to a finite duration input pulse is illustrated.

We can apply Kirchhoff's law to the circuit to find how the input and output signals are related. Assume the current is flowing in a clockwise direction. Then the current flowing into point b is just the potential difference across the resistor, divided by the resistance, $(V_a - V_b)/R$. The current flowing away from point b is the current that flows through the capacitor. Recall that in a capacitor the current is equal to the capacitance, C , multiplying the time rate of change of the potential difference,

$$I_{\text{through capacitor}} = C \frac{d(V_b - V_c)}{dt}. \quad (6.60)$$

The continuity of the current thus tells us that

$$\frac{V_a - V_b}{R} = C \frac{d(V_b - V_c)}{dt}. \quad (6.61)$$

For convenience we can set $V_c = 0$. Then V_a is simply the input voltage, V_{in} , V_b is the output voltage, V_{out} , and Equation (6.61) becomes

$$\frac{V_{in} - V_{out}}{R} = C \frac{dV_{out}}{dt}, \quad (6.62)$$

or

$$\frac{dV_{out}}{dt} + \frac{V_{out}}{RC} = \frac{V_{in}}{RC}. \quad (6.63)$$

This is a linear first-order differential equation, and as such has the analytic solution

$$V_{out}(t) = e^{-t/RC} \left[\int_{-\infty}^t e^{\tau/RC} V_{in}(\tau) d\tau + C_1 \right], \quad (6.64)$$

where C_1 is a constant of integration, chosen so that the solution satisfies the initial conditions. Let's consider an example: How does this circuit respond to a unit impulse like $V_{in}(t) = \delta(t)$? Performing the integration, we find

$$V_{out}(t) = \begin{cases} 0, & t < 0 \\ \frac{1}{RC} e^{-t/RC}, & t \geq 0. \end{cases} \quad (6.65)$$

But what does all this have to do with convolution? Imagine that there is a string of pulses, arriving one after another in rapid succession. The output at any given instant will then be a combination of the responses to the individual input pulses — since the system is *linear*, the combination is simply a sum! That is, the output at a given instant is equal to the sum of the responses to the pulses that have arrived at all previous times. This situation is portrayed in Figure 6.5.

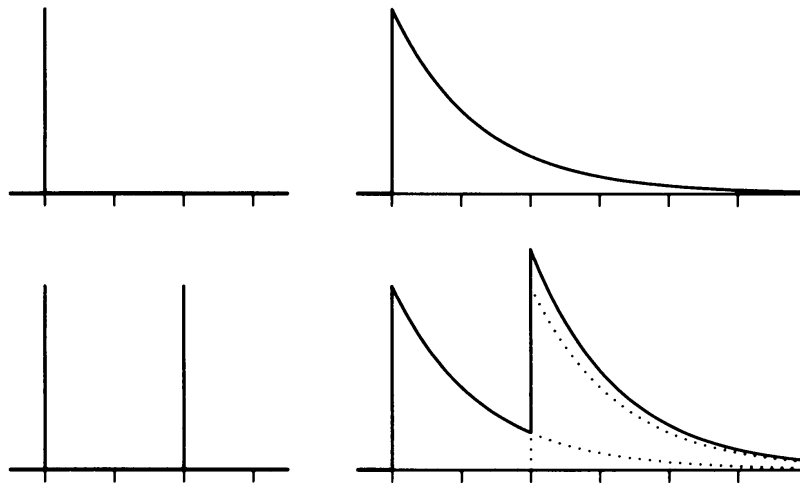


FIGURE 6.5 Response of the RC circuit to delta function inputs. As the delta functions arrive closer in time, the output becomes complicated.

Of course, as we consider more and more pulses arriving at smaller and smaller time separations, we pass to the continuous case. In fact, we can

always write a continuous function as an integral over delta functions,

$$f(t) = \int_{-\infty}^{\infty} f(\tau)\delta(t - \tau) d\tau. \quad (6.66)$$

Let's let $r(t)$ be the response of the system to the impulse. (In our circuit, $r(t) = e^{-t/RC}$.) Then, if the input voltage is written as

$$V_{in}(t) = \int_{-\infty}^{\infty} V_{in}(\tau)\delta(t - \tau) d\tau, \quad (6.67)$$

the output voltage must simply be

$$\begin{aligned} V_{out}(t) &= \int_{-\infty}^{\infty} V_{in}(\tau)r(t - \tau) d\tau \\ &= V_{in} \otimes r. \end{aligned} \quad (6.68)$$

Thus the output of the system is the *convolution* of the input signal with the response of the system to a delta function input. As illustrated in Figures 6.4 and 6.5, the detection tends to broaden features and to smear the input signal — this physical distortion of the input is mathematically described as the convolution.

EXERCISE 6.6

Determine the response, e.g., output, of the RC circuit to the input signal $V_{in}(t) = \cos \gamma t$ by solving the differential equation given in Equation (6.64). Compare your result to that obtained by evaluating the convolution integral of Equation (6.68).

Now consider the Fourier transform of the convolution,

$$\begin{aligned} \mathcal{F}[p \otimes q] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} [p \otimes q] e^{-i\omega t} dt \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p(\tau)q(t - \tau) d\tau \right] e^{-i\omega t} dt \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p(\tau) \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} q(t - \tau) e^{-i\omega t} dt \right] d\tau. \end{aligned} \quad (6.69)$$

With reference to the shifting property, Equation (6.47), we recognize that the term in the square brackets is

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} q(t - \tau) e^{-i\omega t} dt = e^{-i\omega\tau} Q(\omega), \quad (6.70)$$

where $Q(\omega)$ is the Fourier transform of $q(t)$. We then have

$$\begin{aligned}\mathcal{F}[p \otimes q] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p(\tau) e^{-i\omega\tau} Q(\omega) d\tau \\ &= P(\omega)Q(\omega),\end{aligned}\tag{6.71}$$

where $P(\omega)$ is the Fourier transform of $p(t)$. This is known as the *Fourier Convolution Theorem*, and can be written as

$$\mathcal{F}[p \otimes q] = \mathcal{F}[p] \mathcal{F}[q].\tag{6.72}$$

While there are instances where we want the convolution, we are often more interested in the *deconvolution*. That is, if we know $V_{out}(t)$ and $r(t)$, can we find $V_{in}(t)$? If the physical situation is simple enough, the differential equations describing the system can be solved to find V_{in} , but more commonly the equations are not easily solved and such a direct determination is not possible. We can, however, use the convolution theorem. Having previously determined that the response of a system is given as the convolution

$$V_{out} = V_{in} \otimes r,\tag{6.73}$$

we now find that

$$\mathcal{F}[V_{out}] = \mathcal{F}[V_{in} \otimes r] = \mathcal{F}[V_{in}] \mathcal{F}[r].\tag{6.74}$$

Solving for $\mathcal{F}[V_{in}]$, we find

$$\mathcal{F}[V_{in}] = \frac{\mathcal{F}[V_{out}]}{\mathcal{F}[r]}\tag{6.75}$$

or

$$V_{in}(t) = \mathcal{F}^{-1} \left[\frac{\mathcal{F}[V_{out}]}{\mathcal{F}[r]} \right].\tag{6.76}$$

EXERCISE 6.7

Using $r(t)$ and $V_{out}(t)$ from the previous exercise, use the convolution theorem to “determine” $V_{in}(t)$. (What would happen if $\mathcal{F}[r]$ were zero for some ω ?)

While the usual application of convolution is the interaction of a signal with an instrument, *correlation* provides a measure of how much one signal

is similar to another. Mathematically, it's defined as

$$p \odot q = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p^*(\tau) q(t + \tau) d\tau. \quad (6.77)$$

We note that *correlation* involves a complex conjugation of one of the functions, while *convolution* does not. If the two functions are entirely different and unrelated, they are said to be *uncorrelated*.

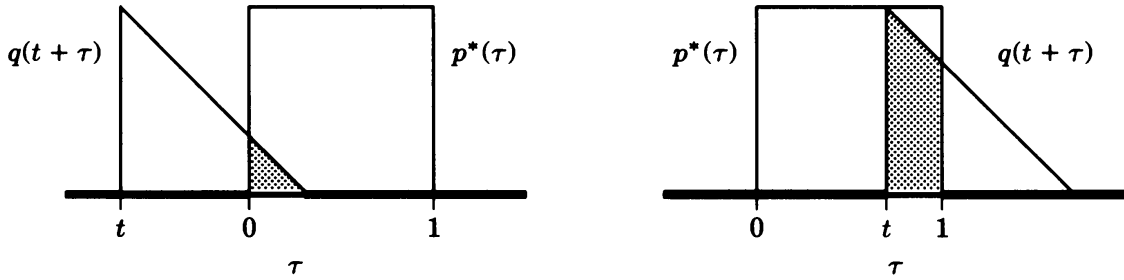


FIGURE 6.6 Graphical representation of correlation. The function q is shifted a time t relative to p^* and the product of the two functions is integrated over all τ to yield the correlation at time t .

Consider the two functions

$$p(t) = \begin{cases} 0, & t < 0, \\ 1, & 0 < t < 1, \\ 0, & t > 1, \end{cases} \quad \text{and} \quad q(t) = \begin{cases} 0, & t < 0, \\ 1 - t, & 0 < t < 1, \\ 0, & t > 1. \end{cases} \quad (6.78)$$

To evaluate the correlation, q is displaced a distance t relative to p , and their product integrated over all τ . This operation is shown in Figure 6.6, in which the evaluation of the correlation at two different times is depicted. Upon evaluating the integral, we find

$$\begin{aligned} p \odot q &= \frac{1}{\sqrt{2\pi}} \int_{\max\{0,t\}}^{\min\{1,1+t\}} (1 - t - \tau) d\tau \\ &= \begin{cases} 0, & t < -1 \\ \frac{1 - 2t - 3t^2}{2\sqrt{2\pi}}, & -1 \leq t \leq 0 \\ \frac{1 - 4t + 3t^2}{2\sqrt{2\pi}}, & 0 \leq t \leq 1 \\ 0, & t > 1. \end{cases} \end{aligned} \quad (6.79)$$

Unlike convolution, for which the order of the functions is immaterial, correlation depends upon the order — $p \odot q$ is not the same as $q \odot p$.

EXERCISE 6.8

Evaluate the correlation $q \odot p$ of the functions described.

In many instances the functions being correlated are not of finite duration. In this case, the integrands don't vanish as $t \rightarrow \pm\infty$, and the integral of Equation (6.77) need not exist. In these situations we define an *average correlation function* as

$$[p \odot q]_{\text{average}} = \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} p^*(\tau) q(t + \tau) d\tau. \quad (6.80)$$

In the particular case that p and q are periodic with period T_0 , this reduces to

$$[p \odot q]_{\text{average}} = \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} p^*(\tau) q(t + \tau) d\tau. \quad (6.81)$$

Let's write the functions $p(\tau)$ and $q(\tau)$ as

$$p(\tau) = \langle p \rangle + \delta_p(\tau) \quad \text{and} \quad q(\tau) = \langle q \rangle + \delta_q(\tau), \quad (6.82)$$

where $\langle p \rangle$ and $\langle q \rangle$ are the mean values of the functions and δ_p and δ_q are the deviations of the functions from their means. Then the correlation of the two functions is

$$\begin{aligned} p \odot q &= \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} [\langle p \rangle + \delta_p(\tau)] [\langle q \rangle + \delta_q(\tau)] d\tau \\ &= \langle p \rangle \langle q \rangle + \langle p \rangle \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} \delta_q(t + \tau) d\tau + \langle q \rangle \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} \delta_p(\tau) d\tau \\ &\quad + \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} \delta_p(\tau) \delta_q(t + \tau) d\tau. \end{aligned} \quad (6.83)$$

Since δ_q has been defined as the deviation of q from its mean value, the integral

$$\lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} \delta_q(t + \tau) d\tau$$

must be zero, and likewise for the integral of δ_p . Let's now specify that the functions are (in some sense) independent of one another — in particular, that the variations in p are unrelated to the variations in q . Then the remaining integral is

$$\lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T/2}^{T/2} \delta_p(\tau) \delta_q(t + \tau) d\tau = 0. \quad (6.84)$$

This specification serves to define what we mean by two functions being *uncorrelated*, in which case we have

$$p \odot q = \langle p \rangle \langle q \rangle. \quad (6.85)$$

As a consequence, if the mean value of either p or q is zero, then the correlation will also be zero.

Of particular interest is the correlation of a function with itself, the *autocorrelation*. Consider, for example, the “box” function

$$p(t) = \begin{cases} 0, & t < 0 \\ 1, & 0 \leq t \leq 1 \\ 0, & t > 1. \end{cases} \quad (6.86)$$

EXERCISE 6.9

Calculate the autocorrelation function of $p(t)$.

If a function is periodic, then it should exhibit a pronounced autocorrelation, since shifting the function by one period will simply shift the function onto itself. For example, the autocorrelation of $\sin t$ is

$$\frac{1}{2\pi} \int_{-\pi}^{\pi} \sin \tau \sin(t + \tau) d\tau = \frac{1}{2} \cos t. \quad (6.87)$$

The autocorrelation of a periodic function thus has oscillations in it, the various maxima occurring whenever the function has been shifted in time by an integral number of periods.

Another example of particular interest to us is at the far extreme from periodic functions — functions that are “random.” There are numerous sources of random noise in real experiments, all characterized as being uncorrelated, at least for sufficiently long times. (All signals relating to real physical situations are continuous and so are necessarily correlated for short

times. As larger time increments are considered, the extent of correlation diminishes.) For long times, then,

$$p \odot p = |\langle p \rangle|^2. \quad (6.88)$$

A rather common occurrence is the presence of a random signal, e.g., *noise*, superimposed upon a signal that we're trying to detect. That is, we might want to observe the signal $s(t)$, but we are actually measuring $p(t) = s(t) + n(t)$, where $n(t)$ is random noise. If the *signal-to-noise ratio* is unfavorable, it might be very difficult to observe the signal. For example, the detected signal might appear as in Figure 6.7. While a certain periodicity to the signal can clearly be discerned, the random scatter of the data makes straightforward analysis difficult.

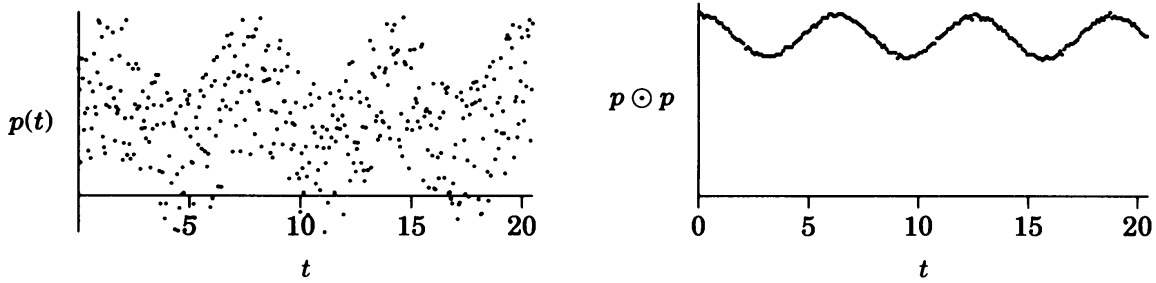


FIGURE 6.7 On the left, a scatter plot of “typical” noisy data. On the right, its autocorrelation.

In contrast, let's evaluate the autocorrelation function,

$$p \odot p = s \odot s + s \odot n + n \odot s + n \odot n. \quad (6.89)$$

Since the signal and the noise are uncorrelated, we have

$$s \odot n = n \odot s = \langle s \rangle \langle n \rangle. \quad (6.90)$$

Thus, for long times we find

$$p \odot p = s \odot s + 2\langle s \rangle \langle n \rangle + |\langle n(t) \rangle|^2, \quad (6.91)$$

so that the autocorrelation of the signal, $s \odot s$, is found on a constant background due to the noise. For example, if $s(t) = \sin t$ then

$$p \odot p = \frac{1}{2} \cos t + |\langle n(t) \rangle|^2. \quad (6.92)$$

This autocorrelation function is also plotted in Figure 6.7. While the noise present in $p(t)$ masks its sinusoidal identity, the autocorrelation clearly exhibits the periodicity in the signal.

The data presented in this figure were simulated, i.e., generated by the computer, by the code

```

delta_time = ...
do i = 1, ...
    time = time + delta_time
    call random (xxx)
    p(i) = sin(time) + 4.d0*xxx
end do

```

This fills the array p with data that ranges from 0 to 5. (Using a uniform distribution of random numbers, such as that provided by the subroutine RANDOM, is not particularly realistic, but suffices for our illustration. A Gaussian distribution of random numbers would be more appropriate in simulating actual experiments.) The autocorrelation function is then approximated by the integral

$$p \odot p \approx \frac{1}{T} \int_0^T p^*(\tau) p(\tau + t) d\tau, \quad (6.93)$$

which we computed by a simple trapezoid rule. The autocorrelation function is then evaluated by the following code:

```

*
*   Compute the autocorrelation for different 'lag' times
*
    do shift = 1, ...
        lag_time = shift * delta_time
*
*   For this particular 'lag_time':
*
        sum = 0.d0
        do i = 1, length + 1
            sum = sum + p(i) * p(i+shift) * delta_time
        end do
        duration = length * delta_time
        auto(i) = sum / duration
    end do

```

For Figure 6.7, `delta_time` was taken to be 0.05 seconds and `length` to be 200 time steps. Changing either of these parameters will modify the actual values obtained but will have little qualitative effect.

Correlation is included in our discussion of Fourier analysis because of the unique character of its Fourier transform,

$$\begin{aligned}
 \mathcal{F}[p \odot q] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} [p \odot q] e^{-i\omega t} dt \\
 &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p^*(\tau) q(t + \tau) d\tau \right] e^{-i\omega t} dt \\
 &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p^*(\tau) \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} q(t + \tau) e^{-i\omega t} dt \right] d\tau \\
 &= Q(\omega) \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p(\tau) e^{-i\omega \tau} d\tau \right]^* \\
 &= P^*(\omega) Q(\omega),
 \end{aligned} \tag{6.94}$$

where $Q(\omega)$ and $P(\omega)$ are the Fourier transforms of $q(t)$ and $p(t)$, respectively. In the particular case of the *autocorrelation*, we find

$$\mathcal{F}[p \odot p] = |P(\omega)|^2. \tag{6.95}$$

We recognize the right side of this equation as the power spectrum — Equation (6.95) is the *Wiener–Khinchine theorem*: the Fourier transform of the autocorrelation function is the power spectrum.

A final word of warning: convolution and correlation are extremely important operations in very diverse physical situations. As a result, different definitions and conventions have arisen in different areas. For example, there is no single, universally accepted symbol representing either convolution or correlation. The ubiquitous factor of 2π , or perhaps $\sqrt{2\pi}$, wanders about. And in some applications, the signals being investigated are always real, so that the complex conjugation is superfluous, and not used — unfortunately, this leads to the possibility of a definition, perfectly appropriate in one discipline, being used inappropriately outside that particular area. There is even disagreement on basic terminology: energy spectrum, power spectrum, power spectral density, and so on, are all used to refer to the same entity. When working in a particular area, you should of course use the standards and definitions of that area — but be aware that they may be slightly different from what you first learned.

Ranging

The use of correlation is important to many areas of experimental physics and has significant technological applications. One such example is in SONAR or

RADAR ranging. By measuring the time delay between the transmission of the signal and the reception of its echo, and knowing the speed of the wave, we can determine the distance to the object which reflected the signal. However, the intensity of the returned echo is often quite low — the intensity falls off as $1/r^4$ — and the detected signal will likely be corrupted by extraneous noise. Rather than looking for the echo directly in the received signal, the echo is sought in the correlation between a reference signal (a copy of the original transmitted signal) and the received signal. This *cross-correlation* will be large at the lag time associated with the time duration of the signal to the reflecting object and its return; at other times, the randomness of the noise in the signal should make the correlation integral vanish (or nearly so).

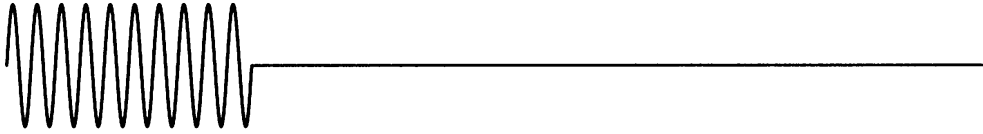


FIGURE 6.8 This is the transmitted signal, as used in the example. The pulse lasts for 10 periods of the sine wave.

As an example, let the signal pulse be a simple sine wave lasting over several cycles, as illustrated in Figure 6.8. We'll simulate the returned signal by adding random numbers, e.g., noise, to the original signal. If the signal is large compared to the noise, then there's little difficulty in detecting the signal and hence determining the distance to the target. But if the noise and the signal are of comparable intensity, the signal is harder to identify. This example is depicted in Figure 6.9, where we've taken the magnitudes of the signal and noise to be the same. Can you see the "signal" in this data? If you look closely, the presence of the signal is barely discernible, but it would be rather difficult to make a reliable distance estimate from the data as they are presented. In contrast, let's consider the cross-correlation between the original reference signal and the echo.

The echo has two components: the original signal, equivalent to the reference $r(t)$, but delayed in time, and the noise $n(t)$. Thus

$$e(t) = \alpha r(t - \Delta) + n(t), \quad (6.96)$$

where α (< 0) represents the attenuation of the signal and Δ is the time delay.



FIGURE 6.9 This is the received signal, sampled at time increments of δ_t . (Since the received signal is much smaller in magnitude than the transmitted signal, the vertical scale of this figure has been exaggerated relative to the scale in Figure 6.8.)

The cross-correlation is then

$$\begin{aligned} r(t) \odot e(t) &= r(t) \odot [\alpha r(t - \Delta) + n(t)] \\ &= \alpha r(t) \odot r(t - \Delta) + \alpha r(t) \odot n(t). \end{aligned} \quad (6.97)$$

But $r(t)$ and $n(t)$ are uncorrelated, and for the particular signal pulse of our example, $\langle r(t) \rangle = 0$. Thus the cross-correlation between the reference signal and its echo is just the autocorrelation of the reference signal, multiplied by α . *The noise itself does not contribute to the cross-correlation!* This is truly amazing — by taking the cross-correlation between the reference signal and the noisy received signal, the noise has been eliminated!

We can easily evaluate the ideal cross-correlation function for our example. We'll take the reference function to be

$$r(t) = \begin{cases} 0, & t < 0, \\ \sin(t), & 0 < t < 20\pi, \\ 0, & t > 20\pi. \end{cases} \quad (6.98)$$

Then the ideal cross-correlation is just a multiple of the autocorrelation function, $r \odot r$, as plotted in Figure 6.10.

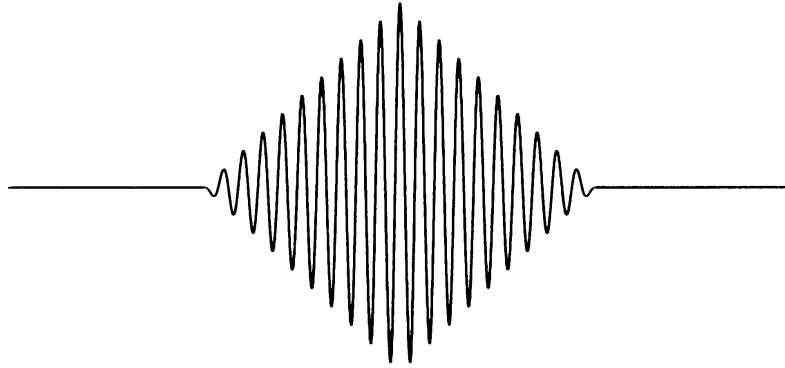


FIGURE 6.10 Ideal cross-correlation between the reference signal and the returned signal, obtained analytically.

In practice, the effects of the noise are usually not totally removed, although the results are still quite dramatic. To illustrate, let's further explore the example we've been considering. Let's store the reference signal in the array `ref(i)` and the echo in `echo(i)`. To evaluate the cross-correlation, we need to evaluate the integral

$$r(t) \odot e(t) = \frac{1}{\sqrt{2\pi}} \int_0^T r(\tau) e(t + \tau) d\tau. \quad (6.99)$$

If this integral were evaluated “perfectly,” then the noise would be totally eliminated. In any real situations, however, the evaluation will be less than exact. Often this integration is done with analog electronic circuitry — basic LRC circuits with appropriately chosen time constants designed so that the output of the circuit is the time integral of the input signal. The integration might also be done digitally, in which case the signals have been sampled at discrete time intervals, such as we are simulating. In either case, integration errors will be introduced. In our example, the integral might be evaluated by the simple trapezoid rule, as in the autocorrelation example discussed earlier. Error is thus introduced by the fundamental inaccuracy of the integration method itself. Statistical error is also introduced, in that the function is being sampled only a finite number of times. For example, we have argued that

$$\frac{1}{T} \int_0^T n(t) dt = 0. \quad (6.100)$$

However, if we evaluate this integral by a finite sum we will find that the integral is not identically zero, although we would expect that the sum approach zero as we take a sufficiently large number of function evaluations. (This, of course, is a statement that you can easily verify for yourself. As more terms are included in the summation, how rapidly does the sum approach zero?)

The cross-correlation obtained for our example problem is displayed in Figure 6.11. Clearly, the noise present in the received signal displayed in Figure 6.9 has been dramatically reduced. Its effect on the cross-correlation is to give a “jitter” to the baseline, a jitter that could be reduced if we worked harder at evaluating the integral.

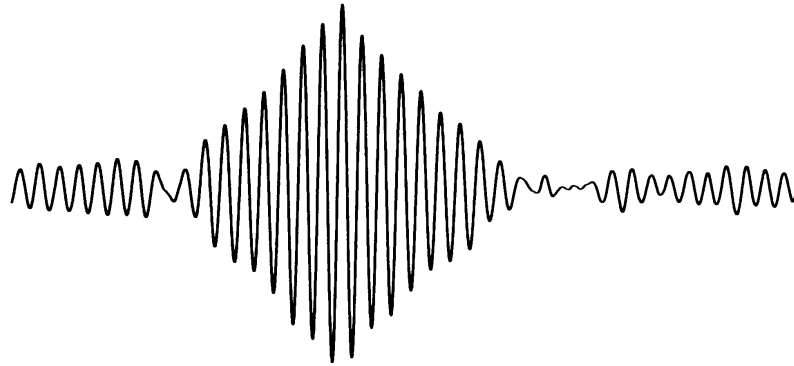


FIGURE 6.11 Calculated cross-correlation between the reference signal and the returned signal.

Although this cross-correlation could certainly be used to determine distance, it would be easier if the correlation were more localized. (This is particularly so if the received signal contains echoes from multiple objects.) This could be done by making the initial signal pulse shorter in duration, but then the energy transmitted — and more importantly, the energy received — would be decreased, while the magnitude of the noise would not be affected. Hence the end result would be to decrease the signal-to-noise ratio. Another approach is to find a reference signal that has a more localized autocorrelation function. The premier example of such a signal is the chirped pulse,

$$r(t) = \sin \omega t, \quad (6.101)$$

in which the frequency ω is itself time-dependent. That is, the frequency changes as a function of time, like a bird’s chirp. The simplest such time variation is a linear one, and we have for example

$$\omega = \omega_0 + ct, \quad (6.102)$$

where c governs how rapidly the frequency changes.

EXERCISE 6.10

Write a computer code to evaluate the cross-correlation of our simulated signal and reference, for a chirped signal pulse. For comparison to our example, take ω_0 to be 1, and investigate the correlation for values of c in the range $0 < c < 0.05$.

An important reason for wanting the autocorrelation function to be as localized as possible is to acquire the ability to discriminate between two different objects. That is, the received signal might contain echoes from two or more reflectors, not just one. If the correlation is broad, then the correlation due to one reflector can overlap the correlation due to the other, hiding the fact that there are two reflectors. By having a localized autocorrelation signature, echoes from objects that are located close to one another will generate correlations that don't overlap, so that the objects can more easily be resolved. If the autocorrelation is sufficiently narrow, different reflectors can be resolved even though the echoes themselves overlap in time.

EXERCISE 6.11

Add a second reflector to the simulation, and investigate the ease with which it is resolved as a function of c .

The Discrete Fourier Transform

Now, let's imagine that we have a physical quantity that's a function of time, and that we measure that quantity in increments Δt . As a result, we have $f(m\Delta t)$, $m = 0, 1, \dots, N - 1$. The Fourier transform $g(\omega)$ is given by the integral

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt. \quad (6.103)$$

Since we have the measurements $f(m\Delta t)$, the integral can be performed numerically, by the trapezoid rule, for example. But there are some problems. First, we didn't take any data points before we started taking data. That is, *we don't have data before $t=0$!* And we don't have *continuous* data, but only data at the times $m\Delta t$! Oops. Maybe this isn't going to be so easy, after all.

Under these conditions, *we cannot calculate the (true) Fourier transform* — we simply don't have enough to work with! All is not lost, however — we can calculate something that *resembles* the Fourier transform and is extremely useful in many cases. We'll assume that we took data for a sufficiently long time T that all the interesting behavior is contained in the data available. Recalling how we developed the Fourier transform from the Fourier series, we'll retrace the steps of our reasoning to develop a discrete representation of the data. That is, we'll *approximate* the Fourier transform of the true data over an infinite range by something like a Fourier series representation of the actual data, on the interval $0 < t < T$.

The complex representation of the Fourier series on the interval $0 <$

$t < T$ can be written as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{i2\pi nt/T}, \quad (6.104)$$

with the coefficients given by the integral

$$c_n = \frac{1}{T} \int_0^T f(t) e^{-i2\pi nt/T} dt. \quad (6.105)$$

Note that this representation of the function is periodic with period T . With the goal of making this look more like a Fourier transform, let's define

$$\Delta\omega = \frac{2\pi}{T}. \quad (6.106)$$

We then approximate the integral by the trapezoid rule, and define the *discrete Fourier transform* as

$$g(n\Delta\omega) = \sum_{m=0}^{N-1} f(m\Delta t) e^{-in\Delta\omega m\Delta t} = \sum_{m=0}^{N-1} f(m\Delta t) e^{-i2\pi mn/N}. \quad (6.107)$$

Since we only have N known quantities, the data taken at N times, we can only determine the transform at N frequencies. Since the $\Delta\omega$ is fixed, the largest frequency we can consider is $(N-1)\Delta\omega$. (Actually, we can only consider frequencies half this large — more on this later.) So the DFT will fail, i.e., give a poor representation, for any function that actually possesses these higher frequencies.

Evaluating the inverse transform is not trivial. Since we have only N frequencies to work with, the summation in the inverse transform can contain only N terms. But it's not obvious that simply truncating the summation of Equation (6.104) is appropriate, or that to do so would not introduce significant error — after all, the trapezoid integration is only accurate to $O(h)$! But in fact, we can find an exact inversion procedure based on the idea of orthogonality. We are acquainted with the idea that functions can be orthogonal to one another, in the sense of performing an integration. They can also be orthogonal in the sense of a summation!

Consider the sum

$$S_N = \sum_{k=0}^{N-1} e^{ik\alpha}. \quad (6.108)$$

If $\alpha = 0$, then every term in the sum is 1 and $S_N = N$. But what if $\alpha \neq 0$? To evaluate the sum, consider the geometric sequence $1, r, r^2, \dots$. The sum of the first N terms is

$$\sum_{k=0}^{N-1} r^k = r^0 + r^1 + \dots + r^{N-1} = \frac{1 - r^N}{1 - r}. \quad (6.109)$$

But if we let $r = e^{i\alpha}$, this is just

$$\sum_{k=0}^{N-1} e^{ik\alpha} = \frac{1 - e^{i\alpha N}}{1 - e^{i\alpha}} = S_N. \quad (6.110)$$

In order to generate an orthogonality relation, we need to find a way to force S_N to be zero if $\alpha \neq 0$. We can do this by simply requiring that

$$e^{i\alpha N} = 1,$$

or that

$$\alpha = 2\pi l/N \quad (6.111)$$

with l an integer. This makes $1 - e^{i\alpha N}$, and hence S_N , zero. We can then write

$$\sum_{k=0}^{N-1} e^{i2\pi kl/N} = \begin{cases} N, & l = 0, \\ 0, & l \neq 0. \end{cases} \quad (6.112)$$

We now express l as the difference between the two integers m and n , and find our orthogonality relation

$$\sum_{k=0}^{N-1} e^{i2\pi km/N} e^{-i2\pi kn/N} = N\delta_{m,n}. \quad (6.113)$$

Returning to the DFT given by Equation (6.107), we multiply both

sides by $e^{i2\pi kn/N}$ and sum over n to find

$$\begin{aligned}
 \sum_{n=0}^{N-1} g(n\Delta\omega) e^{i2\pi kn/N} &= \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} f(m\Delta t) e^{-i2\pi mn/N} e^{i2\pi kn/N} \\
 &= \sum_{m=0}^{N-1} f(m\Delta t) \sum_{n=0}^{N-1} e^{-i2\pi mn/N} e^{i2\pi kn/N} \\
 &= \sum_{m=0}^{N-1} f(m\Delta t) N \delta_{k,m} \\
 &= N f(k\Delta t).
 \end{aligned} \tag{6.114}$$

The *inverse discrete Fourier transform* is then given as

$$f(m\Delta t) = \frac{1}{N} \sum_{n=0}^{N-1} g(n\Delta\omega) e^{i2\pi mn/N}. \tag{6.115}$$

The similarity between the discrete Fourier transforms and the Fourier series or Fourier transforms is unmistakable. However, they are not identical; for example, the DFT uses only finite summations in its evaluations, albeit with particular times and frequencies. These three relations should be thought of as being distinct, although they clearly share a common ancestry.

The Fast Fourier Transform

Although the discrete Fourier transform has considerable computational advantages over the Fourier transform, it is still a computationally intensive operation. With N data points, there will be on the order of N operations performed in each summation. And, of course, this only yields one data point in ω -space. To evaluate all the $g(m\Delta\omega)$, we'll need to perform N summations, for a total of N^2 operations. So, if we double the number of points, we quadruple the effort necessary to perform the calculation. And should we move to two dimensions, the effort goes to order N^4 . Don't ask about three dimensions.

Fourier analysis is a powerful tool in mathematical physics, but as we've indicated, it's very intensive computationally. Its present status as a premier tool of the computational physicist is due to the existence of a streamlined calculational procedure, the *fast Fourier transform*. This algorithm has apparently been independently discovered, or rediscovered, by many investigators, but Cooley and Tukey are generally credited for the discovery, being the first to discuss the algorithm in detail and to bring it to the attention of

the general scientific community. Perhaps the clearest explanation of the algorithm, however, is due to Danielson and Lanczos. If N is an even number, then we can write the DFT as a sum over even-numbered points and a sum over odd-numbered points:

$$\begin{aligned}
 g(n\Delta\omega) &= \sum_{m=0}^{N-1} f(m\Delta t) e^{-i2\pi mn/N} \\
 &= \sum_{m=0, \text{even}}^{N-1} f(m\Delta t) e^{-i2\pi mn/N} + \sum_{m=0, \text{odd}}^{N-1} f(m\Delta t) e^{-i2\pi mn/N} \\
 &= \sum_{j=0}^{N/2-1} f(2j\Delta t) e^{-i2\pi 2jn/N} + \sum_{j=0}^{N/2-1} f((2j+1)\Delta t) e^{-i2\pi (2j+1)n/N},
 \end{aligned} \tag{6.116}$$

where we've let $m = 2j$ in the first term (even points), and $m = 2j + 1$ in the second term (odd points). But this is simply

$$\begin{aligned}
 g(n\Delta\omega) &= \sum_{j=0}^{N/2-1} f(2j\Delta t) e^{-i2\pi jn/(N/2)} \\
 &\quad + e^{-i2\pi n/N} \sum_{j=0}^{N/2-1} f((2j+1)\Delta t) e^{-i2\pi jn/(N/2)} \\
 &= g_{\text{even}}(n\Delta\omega) + e^{-i2\pi n/N} g_{\text{odd}}(n\Delta\omega),
 \end{aligned} \tag{6.117}$$

where we've recognized that the sums are themselves DFTs, with half as many points, over the original even- and odd-numbered points. The original calculation of the DFT was to take on the order of N^2 operations, but this decomposition shows that it actually only requires $2 \times (N/2)^2$ operations! But there's no reason to stop here — each of these DFTs can be decomposed into even and odd points, and so on, as long as they each contain an even number of points. Let's say that $N = 2^k$ — then after k steps, there will be N transforms to be evaluated, each containing only one point! The total operation count is thus *not* on the order of N^2 , but rather on the order of $N \log_2 N$! The fast Fourier transform is *fast*!

The coding of the FFT can be a little complicated, due to the even/odd interweaving that must be done. Since there's little to be gained by such an exercise, we simply present some code for your use.

* Paul L. DeVries, Department of Physics, Miami University

```

*
* Just a little program to check our FFT.
*
*                               start date:  March, 1965
*
* Type declarations
*
      Complex*16 a(8)
      Integer i, k, inv, N
      k = 3
      N = 2**k
      DO i = 1, N
        a(i) = 0
      END DO
      a(3) = 1

      inv = 0
      call FFT( a, k, inv )
      write(*,*)a
      end

*
      Subroutine FFT(A,m,INV)
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine performs the Fast Fourier Transform by
* the method of Cooley and Tukey --- the FORTRAN code was
* adapted from
*
*   Cooley, Lewis, and Welch, IEEE Transactions E-12
*   (March 1965).
*
* The array A contains the complex data to be transformed,
* 'm' is log2(N), and INV is an index = 1 if the inverse
* transform is to be computed. (The forward transform is
* evaluated if INV is not = 1.)
*
*                               start:  1965
*                               last modified:  1993
*
      Complex*16 A(1), u, w, t
      Double precision ang, pi
      Integer N, Nd2, i, j, k, l, le, le1, ip
      Parameter (pi = 3.141592653589793d0)

```

```

*
* This routine computes the Fast Fourier Transform of the
* input data and returns it in the same array. Note that
* the k's and x's are related in the following way:
*
* IF      K = range of k's      and      X = range of x's
*
* THEN  delta-k = 2 pi / X      and      delta-x = 2 pi / K
*
* When the transform is evaluated, it is assumed that the
* input data is periodic. The output is therefore periodic
* (you have no choice in this). Thus, the transform is
* periodic in k-space, with the first N/2 points being
* 'most significant'. The second N/2 points are the same
* as the fourier transform at negative k!!! That is,
*
*          FFT(N+1-i) = FFT(-i)   ,i = 1,2,...,N/2
*
      N    = 2**m
      Nd2 = N/2
      j    = 1
      DO i = 1, N-1
        IF( i .lt. j ) THEN
          t    = A(j)
          A(j) = A(i)
          A(i) = t
        ENDIF
        k = Nd2
100      IF( k .lt. j ) THEN
          j = j-k
          k = k/2
          goto 100
        ENDIF
        j = j+k
      END DO
      le = 1
      DO l = 1, m
        le1 = le
        le  = le + le

        u = ( 1.D0, 0.D0 )
        ang = pi / dble(le1)
        W = Dcmplx( cos(ang), -sin(ang) )
        IF(inv .eq. 1) W = Dconjg(W)

```

```

      DO j = 1, le1
        DO i = j, N, le
          ip = i+le1
          t  = A(ip)*u
          A(ip) = A(i)-t
          A(i) = A(i)+t
        END DO
        u=u*w
      END DO

END DO

IF(inv .ne. 1) THEN
  DO i = 1, N
    A(i) = A(i) / dble(N)
  END DO
ENDIF
end

```

EXERCISE 6.12

Verify that this code performs correctly. What should the results be? Note that, as often happens when doing numerical work, the computer might give you a number like 10^{-17} where you had expected to find a zero.

Life in the Fast Lane

The development of the fast Fourier transform is one of the most significant achievements ever made in the field of numerical analysis. This is due to *i)* the fact that the Fourier transform is found in a large number of diverse areas, such as electronics, optics, and quantum mechanics, and hence is extremely important; and *ii)* the FFT is *fast*. For our present purposes, we will consider the sampling of some function at various time steps — the Fourier transform will then be in the frequency domain. Such an analysis is sometimes referred to as being an harmonic analysis, or a function is said to be synthesized (as in electronic music generation.)

Earlier, we noted that while it appeared that the highest frequency used in the DFT (or the FFT) was $(N - 1)\Delta\omega$, the highest frequency actually used is only half that. The reason is quite simple: while $f(t)$ is periodic in time, $g(\omega)$ is periodic in frequency! (This is obvious, after the fact, since the

exponential functions used in the DFT are themselves periodic.) Then the frequencies from $N\Delta\omega/2$ up to $(N-1)\Delta\omega$ are actually the negative frequencies from $-N\Delta\omega/2$ up to $-\Delta\omega$! This frequency, $\omega_{Nyquist} = N\Delta\omega/2$ is called the *Nyquist frequency*, and has a very clear physical basis. Let's imagine that you have a pure cosine function that oscillates twice a second, and that you sample that function every second. What do you see? Remember, you don't know what value the function has except at the times you measure it! So what you see is a *constant* function! How often must you sample the function to get a true picture of how rapidly it's oscillating?

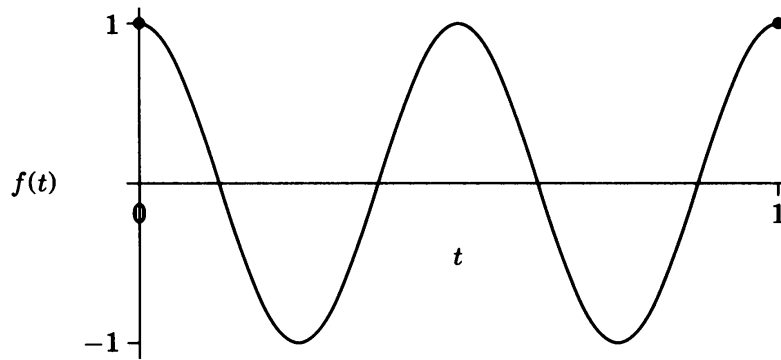


FIGURE 6.12 The function $f(t) = 4\pi t$. Sampled once a second — or even once every half a second — you would think the function were a constant.

To help investigate the properties of the FFT, you should write a subroutine to plot the magnitude of the transform for you. This can be thought of as a spectral decomposition, or spectrum, of the original data. Since frequencies above the Nyquist frequency are actually negative frequencies, I suggest that you plot these as negative frequencies, so that the spectrum is plotted on the interval $-\omega_{Nyquist} < \omega < \omega_{Nyquist}$. You might find it useful to plot the original data and its spectrum on the same screen.

As suggested by this discussion, the rate at which data are taken is extremely important to the proper utilization of the FFT. In particular, the data must be taken fast enough that nothing dramatic happens *between* the sampled times.

EXERCISE 6.13

Consider the function $f(t) = \cos 6\pi t$. Sample the function every second, for 32 seconds, and evaluate the spectrum of the sampled data. (Use $N = 32$ throughout this exercise.) Repeat the exercise, taking data every half-second, for 16 seconds, and then every quarter-second,

etc. What must the sampling rate be for you to determine the true frequency? (Note that $\Delta\omega$ is different for each of the spectra obtained.)

This exercise addresses the question of sampling rate, and in particular, the perils of *undersampling* the data. It also illustrates the artifact of *aliasing* — if a function is undersampled, the high-frequency components don't just go away, but rather they will be *disguised* as low-frequency components! This is the case depicted in Figure 6.12, for example. For the FFT to provide a meaningful representation of the data, the data must be sampled at twice the frequency of the highest component contained in it, or higher. And even if the data are sampled at a sufficiently high rate, there can be problems. The FFT assumes that the data are periodic *on the sampled interval* — and if it's not?

EXERCISE 6.14

Consider the function $f(t) = \cos 3t$. Sample the data every second for 8 seconds. This is a sufficiently high rate that undersampling should not be a problem. Evaluate and display the spectrum. What went wrong?

This problem is termed *leakage*. The actual frequency of the data *is not* one of the $m\Delta\omega$ at which the transform is evaluated. So, the FFT *tries* to describe the data by using nearby frequencies, distributing the magnitude of the transform across several frequencies — that is, the magnitude *leaks* into nearby frequency bins. This problem can be alleviated by increasing the resolution of the frequencies. Since

$$\Delta\omega = \frac{2\pi}{T}, \quad (6.118)$$

this can be accomplished by sampling the data for a longer time. *Note that the sampling rate is not involved here — only the total observation time!* If we are to sample the function at the same rate, which we must do to avoid undersampling, then we must increase the number of times the function is sampled!

EXERCISE 6.15

Repeat the previous exercise, sampling the function $f(t) = \cos 3t$ every second, but sample for 16, 32, and 64 seconds. Compare the spectra obtained. Note that $\omega_{Nyquist}$ is the same for all the spectra.

An alternate way of viewing the problem of leakage is to note that the sampling interval may not be commensurate with the period of the function, as in Figure 6.13. Sampling the function $f(t) = \cos 3t$ once a second always leads to a discontinuity at the end of the sampling period. Now, if you were to

sample at

$$\Delta t = \frac{2\pi}{3N},$$

you *might* get a different result.

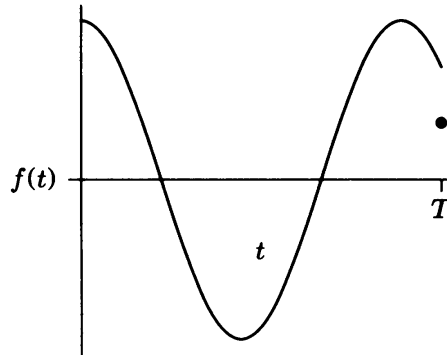


FIGURE 6.13 A function sampled for a period T incommensurate with the actual period of the function. (Since there is a jump discontinuity in the data, the last data point used should be the mean.)

Line spectra, such as we have been investigating, are quite common in physics. And associated with them are the difficulties sometimes encountered when more than one line is present.

EXERCISE 6.16

Consider the function $f(t) = \sin(2 + \alpha)t + \sin(4 - \alpha)t$, as the parameter α varies from 1 down to 0, with $N = 32, 64$, and 128. Notice anything “unusual” as α gets in the 0.7–0.8 range? Be prepared to explain your results.

Spectrum Analysis

The natural application of FFTs, spectrum analysis, and so on, is to experimental data — by converting data obtained as a function of time to data expressed in terms of frequency, new insights can be gained. However, such an analysis can be just as useful in understanding the physical content of *synthetic data*, such as produced by a numerical calculation.

As an example, let’s reconsider the Van der Pol oscillator. Previously, we saw that a phase space diagram was a useful tool in understanding the

oscillator's behavior. Now, we'll see that Fourier analysis, via the FFT, will complement that understanding.

The FFT requires data at fixed time increments. We could use the Runge–Kutta–Fehlberg integrator and interpolate to obtain the desired data. Of course, the interpolation would introduce its own error, which seems to subvert our intent in designing the RKF integrator in the first place. So, let's modify the Runge–Kutta–Fehlberg subroutine so that the integrator provides the desired data, directly. Let's introduce the variable TARGET, which is the *next* point at which the function is desired. Then, after the next step size has been predicted, we simply inquire if this step takes us past TARGET. If it doesn't, nothing changes; if it does, we define the step size to be exactly the difference between the target and the current step. The modifications to the code then look like this:

```

...
*
* The solution vectors are evaluated at precisely the
* TARGET, and stored in the array DATA for subsequent use.
*
      complex*16 data(1024)
      double precision TARGET, delta
      integer ifft, NFFT
      ...
      ifft = 0
      nfft = 512
      TARGET = 100.0d0
      delta = 0.1d0
      ...
      n = 2
100    call der(x0,y0,f0)
200    x = x0 + a1*H
      ...
      Hold = H
      h = 0.9d0 * h * (MaxErr/BigErr)**0.2d0

      IF( BigErr .gt. MaxErr ) goto 200
      x0 = x0 + hold
*
* The solution has been accepted, and the "current
* position" updated. "H" is the NEXT step to be taken.
* First, check if X0 is equal to TARGET (or close enough):
*
      IF( dabs((X0-TARGET)/X0) .lt. 1.d-5 )THEN

```

```

* TARGET has been hit --- increment the counter IFFT,
* store the data, and move the TARGET:
*
      ifft = ifft + 1
      data(ifft) = yhat(1)
      TARGET = TARGET + delta
    ENDIF
*
* Second, if the next step takes us past the TARGET,
* redefine H:
*
      IF( h .gt. TARGET-X0 ) h = TARGET-X0
      DO i = 1, n
        y0(i) = yhat(i)
      END DO
      ...
*
* Have we gone far enough? Collected enough data?
*
      IF (ifft .lt. nfft) goto 100
      ...

```

This will fill the array DATA with positions at 512 data points, starting with the position at x (time, that is) = 100, the initial value of TARGET, at increments of $\delta t = 0.1$. Note that we don't start taking data immediately — nonlinear oscillators tend to wander around for a while, depending upon their initial conditions. But we're interested in their long-term behavior, so we wait some period of time before taking data to let the transients disappear. These data are presented in Figure 6.14. Clearly, it's periodic, but not sinusoidal.

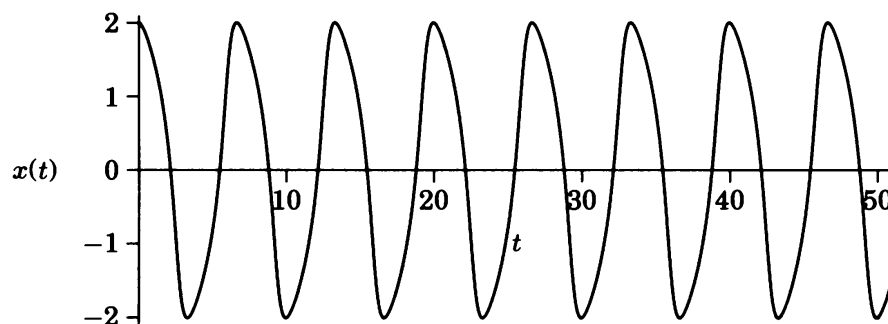


FIGURE 6.14 Position versus time for the Van der Pol oscillator using $\Delta t = 0.1$ seconds and 512 points.

The data, taken in the time domain, can then be Fourier-transformed to the frequency domain. Since the data are real, plotting the positive frequencies is sufficient. The frequency spectrum $|g(\omega)|$ is plotted in Figure 6.15.

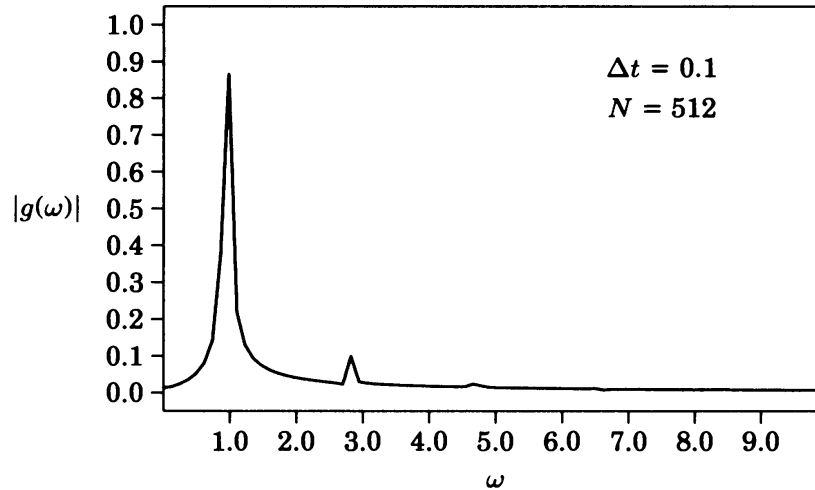


FIGURE 6.15 The frequency spectrum $|g(\omega)|$ of the Van der Pol oscillator, using $\Delta t = 0.1$ seconds and 512 points.

Clearly, there is a main peak near $\omega = 1$, and a second peak a little short of 3. Maybe, if I didn't bump the plotter while it was at work, there might be a third peak between 4.5 and 5, but I'm not sure. One way to enhance the spectrum in order to make small features more visible is to plot the data on a logarithmic scale rather than a linear one. The same data plotted in Figure 6.15 are also presented in Figure 6.16 — not only is the third peak visible, but a 4th and 5th as well! Still, the “background noise” is pretty high, and the resolution is pretty low.

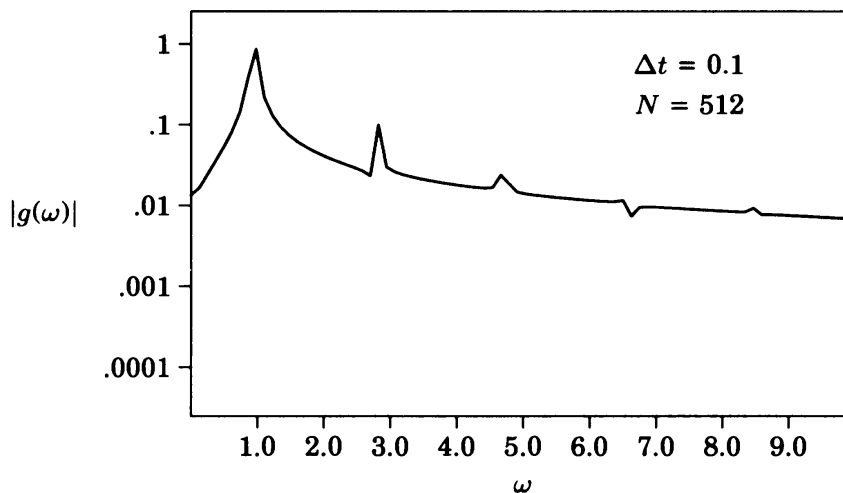


FIGURE 6.16 The same spectrum as in Figure 6.15, but plotted on a logarithmic scale to enhance the smaller features of the graph.

Why are there so many peaks? Recall that the Van der Pol oscillator is very stable, so that its period is very regular. *But it's not a simple harmonic oscillator* — we can see that in Figure 6.14! More than one harmonic of the fundamental frequency is required to describe its relatively complicated motion. And that's what the extra peaks are, the harmonics. Note that the peaks appear at approximately odd integer multiples of the fundamental — but not at even multiples.

If we want to improve the frequency resolution of the spectrum, the total observation time must be increased. This can be done either by increasing Δt , or by increasing N . The Nyquist frequency for $\Delta t = 0.1$ seconds is 31.416 sec^{-1} , but the spectrum shows that the contributions from the higher frequencies are not significant. So we can double Δt , thereby halving the Nyquist frequency, without fear of undersampling, while at the same time doubling the frequency resolution. This new spectrum is plotted in Figure 6.17. The higher harmonics are much more apparent in this figure, and the “background” is somewhat reduced. Overall, the spectrum is better than the previous one. Still, the peak is rather broad — is this *real*, or an artifact of the approach?

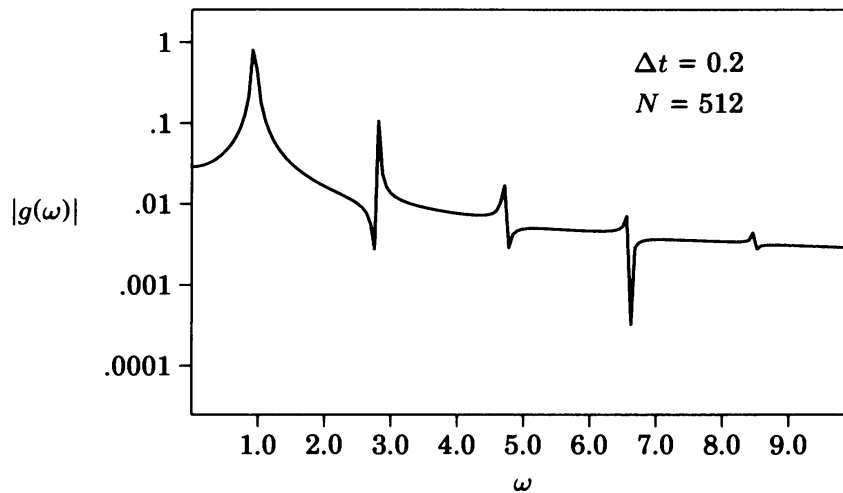


FIGURE 6.17 The frequency spectrum using $\Delta t = 0.2$ seconds, on a logarithmic scale.

Let's reexamine the original time-domain data, as presented in Figure 6.14. We see that the observation time is not an integer number of periods. That is, the sampling period is not commensurate with the period of the oscillator — Δt was chosen at our convenience, and is not related to the physics of the problem. The reason for the broad peaks in these spectra is due to *leakage*. While we could improve the resolution by further increasing the observation time, choosing a commensurate sampling rate will probably have

a greater impact upon the quality of the spectrum. From the data of Figure 6.14, we find that the period of an oscillation is 6.66338 seconds. Thus, if we were to sample at $\Delta t = 0.208231$ seconds, 16 complete cycles of the oscillator would be sampled with 512 data points. Sampling at this rate, we obtain the spectrum of Figure 6.18.

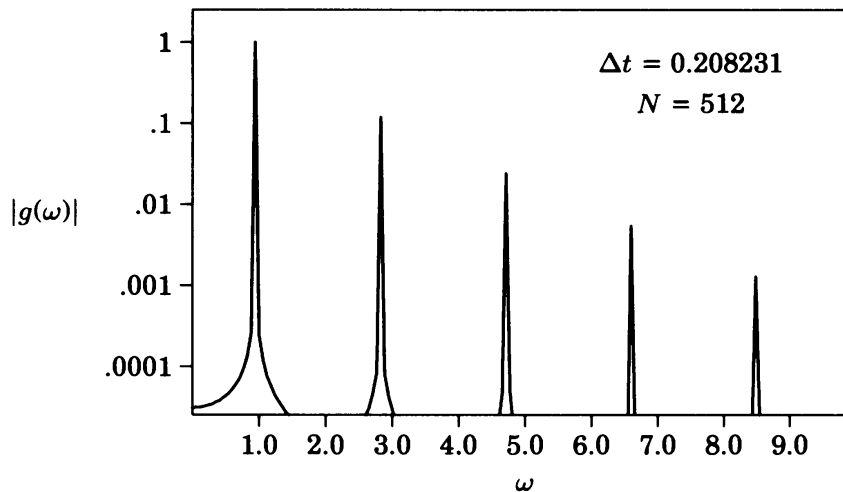


FIGURE 6.18 The frequency spectrum obtained by using a sampling period commensurate with the oscillator.

The difference between this spectrum and the previous ones is astounding. Here the peaks have narrowed to nearly delta functions, and the background has diminished by several orders of magnitude. This spectrum is clearly superior to the one of Figure 6.17, although it was obtained with essentially the same amount of effort — a result of using the knowledge we have to better our understanding. It also suggests that knowledge obtained in the time domain complements that obtained, or obtainable, in the frequency domain. They are, in fact, two different approaches to understanding the same physical processes.

The Duffing Oscillator

The Van der Pol oscillator is actually rather boring — its behavior is pretty staid. In its place, let's consider *Duffing's oscillator*, described by the differential equation

$$\ddot{x} + k\dot{x} + x^3 = B \cos t. \quad (6.119)$$

This is clearly a nonlinear oscillator and, as we shall see, admits a wide range of possible behaviors. In fact, it can be periodic, or its motion can be so complicated as to be seemingly unpredictable. Now, that's a very strange statement

— the motion is perfectly well described by the differential equation. That is, it's certainly *deterministic*. And yet, the motion can seem to be totally random, even *chaotic*. Hmmm ...

■ EXERCISE 6.17

Use our recently developed methods of Fourier analysis to describe the motion of Duffing's oscillator. Note that as the parameters change, the very nature of the motion can change. Use $k = 0.1$, and describe the motion for $B = 2, 4, 6, 8, 10, 12, 14$, and 16 . You might also find phase space diagrams useful.

Computerized Tomography

Fourier analysis is commonplace throughout physics, particularly in areas of “applied” physics. In some areas, such as modern optics, it can be argued that Fourier analysis is truly *fundamental* to the field. Such widespread applicability of Fourier methods certainly warrants its study in computational physics, although we cannot begin to devote the space it deserves. Rather, we shall present only a single example — but one that has revolutionized modern medicine.

When Wilhelm Roentgen discovered x-rays in 1895, its importance to the practice of medicine was quickly recognized. Within months, the first medical x-ray pictures were taken. For his efforts, Roentgen was awarded the first Nobel Prize in Physics in 1901. In the 1960s and 1970s, another revolution occurred as x-rays were employed to obtain detailed internal images of the body, using computerized tomography. Today, CAT scans are an important and commonly used tool in the practice of medicine. In 1979, Cormack and Hounsfield were awarded the Nobel Prize in Medicine for their efforts. Our interest in computerized tomography is simple — CT is fundamentally based upon Fourier transforms.

As a beam of x-rays passes through a uniform slab of material, the intensity of the beam decreases in a regular pattern. If we measure that intensity, we find that the intensity to be

$$I = I_0 e^{-\lambda d}, \quad (6.120)$$

where I_0 is the initial intensity, d is the thickness of the slab, and λ is an appropriate coefficient. λ is often called an absorption coefficient, but there are actually many processes occurring that diminish the intensity of the beam.

In particular, the detector is arranged to measure the intensity of the beam that passes straight through the object, so that any x-rays that are *scattered* will not be detected. (Furthermore, the detector is constructed so that x-rays entering at an angle, and hence scattered from some other region of the object, are rejected.) The reduction of the signal in CT is primarily due to scattering out of the forward direction. We'll refer to the loss of intensity, to whatever cause, as *attenuation*.

In CT, a two-dimensional slice through an object is imaged. In its simplest configuration, x-rays are directed in a series of thin pencil-like parallel beams through the image plane, and the attenuation of each beam measured. Since the object is nonuniform, the attenuation of the beam varies from point to point, and the total attenuation is given as an integral along the path of the beam. That is, if $f(x, y)$ is the two-dimensional attenuation function, the intensity of each of the beams will be of the form

$$I = I_0 e^{-\int f(x, y) d\eta}, \quad (6.121)$$

where $d\eta$ is an element along the path. Each parallel beam, offset a distance ξ from the origin, traverses a different portion of the object and is thus attenuated to a different extent. From the collection of beams a profile of the object can be built. More to the point, the *projection at ϕ* can be obtained by evaluating the logarithm of the measured I/I_0 ratio,

$$p(\xi; \phi) = -\ln \left(\frac{I}{I_0} \right) = \int f(x, y) d\eta. \quad (6.122)$$

Our goal is to reconstruct $f(x, y)$ from the various projections $p(\xi; \phi)$ obtained at different angles ϕ .

Consider the coordinate systems in Figure 6.19. Associated with the object being scanned are the “fixed” xy -coordinates. We have a second coordinate system aligned with the x-ray beams, with coordinate η along the beam path and ξ perpendicular to them. The angle ϕ is simply the angle between $+x$ and $+\xi$ (and between $+y$ and $+\eta$). These coordinates are related by the familiar relations

$$\xi = x \cos \phi + y \sin \phi \quad (6.123)$$

and

$$\eta = -x \sin \phi + y \cos \phi. \quad (6.124)$$

The projection can be thought of as a function of ξ obtained at the particular angle ϕ .

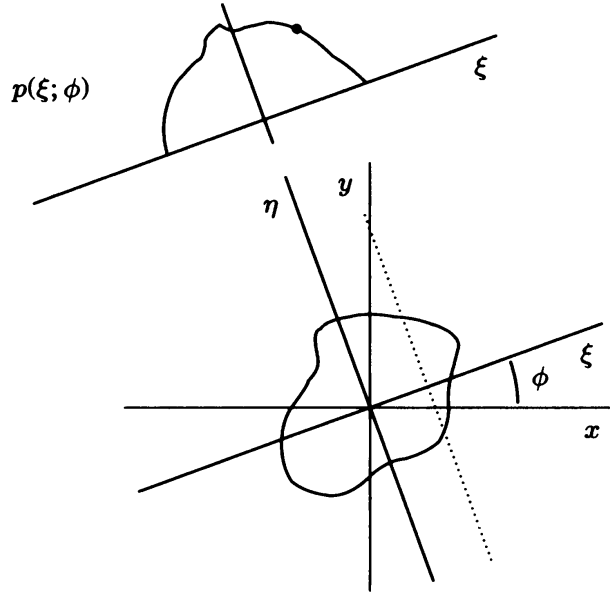


FIGURE 6.19 CAT scan geometry. The path of a typical beam is shown by the dotted line; its attenuation is represented by a single point in the projection $p(\xi; \phi)$.

Let's look at the Fourier transform of $f(x, y)$, which is simply

$$\mathcal{F}[f(x, y)] = F(k_x, k_y) = \frac{1}{2\pi} \iint_{-\infty}^{\infty} f(x, y) e^{i(k_x x + k_y y)} dx dy, \quad (6.125)$$

and write the transform in terms of the $\xi\eta$ -coordinate system. In transform space, k_ξ and k_η are rotated with respect to k_x and k_y in exactly the same fashion as ξ and η are rotated with respect to x and y , as shown in Figure 6.20. Thus

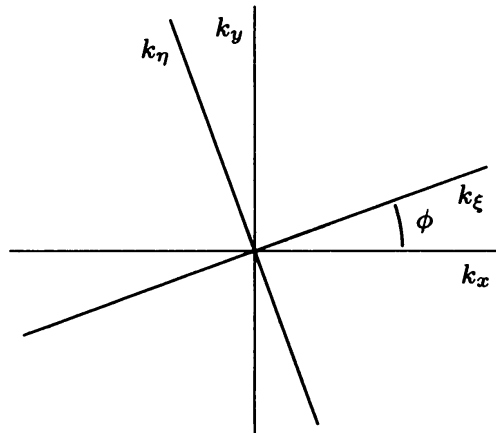


FIGURE 6.20 The relevant coordinate systems in transform space.

$$k_\xi = k_x \cos \phi + k_y \sin \phi \quad (6.126)$$

and

$$k_\eta = -k_x \sin \phi + k_y \cos \phi. \quad (6.127)$$

We easily find that the exponential factor appearing in the Fourier integral can be written as

$$e^{i(k_x x + k_y y)} = e^{i(k_\xi \xi + k_\eta \eta)}. \quad (6.128)$$

To this point, we have been perfectly general. But now we're going to restrict ourselves. First, we fix ϕ in both coordinate and transform space. Then, instead of evaluating the Fourier transform *everywhere* in the $k_x k_y$ -plane, we're only going to evaluate the transform on one line in the plane — the k_ξ axis. (Note that this line is coincident with the radial coordinate of a polar coordinate system, but extends from $-\infty$ to $+\infty$.) Since $k_\eta = 0$ along this line, we have

$$F(k_\xi \cos \phi, k_\xi \sin \phi) = \frac{1}{2\pi} \iint_{-\infty}^{\infty} f(x, y) e^{ik_\xi \xi} d\xi d\eta, \quad (6.129)$$

where we've replaced the element of area $dx dy$ by the equivalent $d\xi d\eta$ in the integration. But with reference to Equation (6.122), we see that *the η integral is simply the projection* $p(\xi; \phi)$, so that

$$F(k_\xi \cos \phi, k_\xi \sin \phi) = \frac{1}{2\pi} \int_{-\infty}^{\infty} p(\xi; \phi) e^{ik_\xi \xi} d\xi. \quad (6.130)$$

This is known as the projection theorem, and simply states that the Fourier transform of the projection at ϕ is the Fourier transform of the attenuation function along ϕ in transform space.

Measuring the projections $p(\xi; \phi)$ at several different angles will yield the transform of the attenuation function throughout the two-dimensional plane, one radial line at a time. In principle, the attenuation function could be obtained by evaluating the inverse Fourier transform. There is, however, a substantial obstacle to this straightforward approach — the points at which the transform is known are not the same as those needed for the inverse transform.

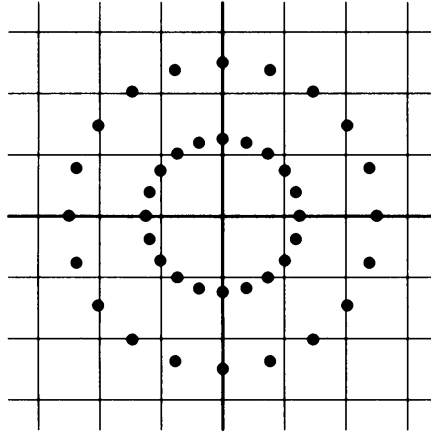


FIGURE 6.21 The Fourier transform is known on the polar grid, indicated by the black dots, while the straightforward FFT requires the data to be known on a Cartesian grid.

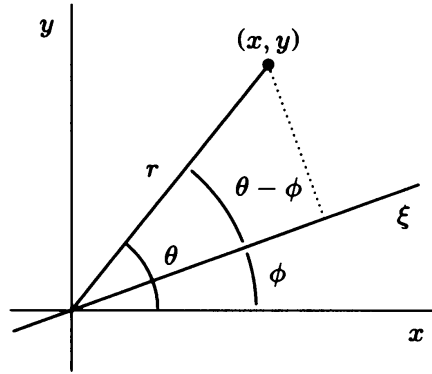
In building the Fourier transform from the projections, knowledge of the transform was acquired along the radial coordinate at different angles, as illustrated in Figure 6.21. But the grid points of a polar coordinate system do not coincide with the grid points of a Cartesian system, although these are precisely where they're needed if the Fourier transform is to be evaluated by the FFT. Increasing the number of grid points in a projection, or the number of projections, does not help solve the problem since the points still will not be where they are needed. Ultimately, interpolation must be used. A high order interpolation will require extensive calculations, in addition to the two-dimensional FFT needed to obtain the attenuation function in coordinate space. Any interpolation introduces additional error, and interpolation in transform space is particularly tricky — the error at one grid point, when transformed back to coordinate space, is spread over the entire plane. The image thus obtained is the sum of the true image plus error contributions from every interpolated point in transform space. As a result of these difficulties, direct inversion is rarely used. Rather, so-called *back projection* methods are employed. In effect, these methods perform the (necessary!) interpolation in *coordinate space* rather than in transform space.

To see how this works, let's begin by writing the attenuation function as

$$f(x, y) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(k_x, k_y) e^{-i(k_x x + k_y y)} dk_x dk_y. \quad (6.131)$$

Writing x and y in terms of the polar coordinates r and θ (see Figure 6.22), and k_x and k_y in terms of ρ and ϕ , we find

$$k_x x + k_y y = \rho \cos \phi r \cos \theta + \rho \sin \phi r \sin \theta = \rho r \cos(\theta - \phi), \quad (6.132)$$

FIGURE 6.22 Geometry relating (x, y) and (r, θ) to ξ .

so that $f(x, y)$ becomes

$$\begin{aligned}
 f(r \cos \theta, r \sin \theta) &= \frac{1}{2\pi} \int_0^\infty \int_0^{2\pi} F(\rho \cos \phi, \rho \sin \phi) e^{-i\rho r \cos(\theta-\phi)} \rho d\phi d\rho \\
 &= \frac{1}{2\pi} \int_0^\infty \int_0^\pi F(\rho \cos \phi, \rho \sin \phi) e^{-i\rho r \cos(\theta-\phi)} \rho d\phi d\rho \\
 &\quad + \frac{1}{2\pi} \int_0^\infty \int_\pi^{2\pi} F(\rho \cos \phi, \rho \sin \phi) e^{-i\rho r \cos(\theta-\phi)} \rho d\phi d\rho \\
 &= \frac{1}{2\pi} \int_0^\infty \int_0^\pi F(\rho \cos \phi, \rho \sin \phi) e^{-i\rho r \cos(\theta-\phi)} \rho d\phi d\rho \\
 &\quad + \frac{1}{2\pi} \int_0^\infty \int_0^\pi F(-\rho \cos \phi, -\rho \sin \phi) e^{+i\rho r \cos(\theta-\phi)} \rho d\phi d\rho \\
 &= \frac{1}{2\pi} \int_0^\pi \left(\int_0^\infty F(\rho \cos \phi, \rho \sin \phi) e^{-i\rho r \cos(\theta-\phi)} \rho d\rho \right. \\
 &\quad \left. + \int_0^\infty F(-\rho \cos \phi, -\rho \sin \phi) e^{+i\rho r \cos(\theta-\phi)} \rho d\rho \right) d\phi.
 \end{aligned} \tag{6.133}$$

In the last integral, we might be tempted to make the substitution $\rho \rightarrow -\rho$. But ρ is the radial polar coordinate and is necessarily positive — to avoid confusion, we should leave it positive. However, if the ϕ integral is done last, the angle is fixed while performing the ρ integral. And at fixed ϕ , k_ξ lies along ρ . Thus, in the next-to-last integral we make the substitution $\rho \rightarrow k_\xi$, while in the last we use $\rho \rightarrow -k_\xi$ and reverse the limits of integration. By our definition

of these coordinate systems, we also have that $r \cos(\theta - \phi) = \xi$. Thus we find

$$\begin{aligned}
 f(r \cos \theta, r \sin \theta) &= \frac{1}{2\pi} \int_0^\pi \left(\int_0^\infty F(k_\xi \cos \phi, k_\xi \sin \phi) e^{-ik_\xi \xi} k_\xi dk_\xi \right. \\
 &\quad \left. + \int_{-\infty}^0 F(k_\xi \cos \phi, k_\xi \sin \phi) e^{-ik_\xi \xi} (-k_\xi) dk_\xi \right) d\phi \\
 &= \frac{1}{2\pi} \int_0^\pi \int_{-\infty}^\infty F(k_\xi \cos \phi, k_\xi \sin \phi) e^{-ik_\xi \xi} |k_\xi| dk_\xi d\phi \\
 &= \int_0^\pi \tilde{p}(\xi; \phi) d\phi,
 \end{aligned} \tag{6.134}$$

where

$$\tilde{p}(\xi; \phi) = \frac{1}{2\pi} \int_{-\infty}^\infty F(k_\xi \cos \phi, k_\xi \sin \phi) e^{-ik_\xi \xi} |k_\xi| dk_\xi \tag{6.135}$$

is the *modified projection at ϕ* .

Let's review what we've done. Taking the Fourier transform of the projections, we know the Fourier transform of the attenuation function on a polar grid in transform space. To recover the function in coordinate space, we need to perform the inverse transform, but this is easily done *only* in Cartesian coordinates. To avoid interpolation in transform space, the inverse transform is written in terms of polar coordinates. Then, instead of integrating from 0 to 2π , we integrate from 0 to π and extend the "radial integration" to negative values. In this way, we're still integrating over the entire two-dimensional plane, but the "radial" integral now has the limits of a Cartesian coordinate. The price we pay is the presence of the factor $|k_\xi|$ in the integrand — but the integral has the form of an inverse Fourier transform and can be evaluated by the FFT!

What, then, are the actual steps to be taken to reconstruct the image? One possible scheme is the following:

0. Initialize everything. Zero the entries of the image. The outermost loop will integrate over the various angles.
1. Obtain the projection $p(\xi; \phi)$ at a new angle ϕ . The projection will be known (measured) at the points ξ_i .
2. Using the FFT, obtain $F(k_\xi \cos \phi, k_\xi \sin \phi)$ via Equation (6.130).

3. Using the inverse FFT, evaluate $\tilde{p}(\xi; \phi)$ via Equation (6.135). This modified projection will be known at the same points ξ_i as the original projection.
4. Evaluate the contribution to the image from this one projection.
 - 4a. For a particular xy coordinate, determine ξ from Equation (6.123).
 - 4b. Approximate the modified projection at ξ by interpolation of the $p(\xi_i; \phi)$. (Linear interpolation is usually sufficient.)
 - 4c. Add this contribution to the ϕ integral to the image array, Equation (6.134).
 - 4d. Repeat steps 4a, 4b, and 4c for all xy coordinates in the image array.
5. Display the image obtained thus far. (Optional.)
6. Repeat steps 1–5 for all angles in the ϕ integration.

With this outline as a guide, construction of a computer program is made easier. But first, we need to obtain the projections. In practice, of course, this is the data that would be obtained from the CT scanning device. For our purposes, the function `CT_PROJECTION(xi,phi)`, which returns the value of the projection at ξ and ϕ , is provided. The test image is an L-shaped object and a circle, as shown in Figure 6.23. (That is, these objects will attenuate any x-ray beams traversing the region.) Note that the entire image is contained within a circle of unit radius.

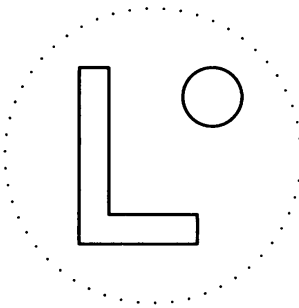


FIGURE 6.23 A test image to explore computerized tomography.

Since the numerical values in the image are obtained by interpolation,

the size of the image is separated from the number of points in the projection. To emphasize this point, we'll make the image 51×51 , a size incommensurate with any of the other parameters of the problem. A sketch of the computer program would look like the following:

Program CT

```

*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* The program CT performs image reconstruction from
* given "x-ray" projections.
*
      double precision image(51,51), xi(32), projection(32)
      double precision pi, phi, k
      complex*16 temporary(32)
      integer log_2_N, N, halfN, num_phi, i, j, count,
+           index, pixels
      parameter ( pi = 3.14159 26535 89793 d0, log_2_N = 5,
+           num_phi=12, pixels = 51 )
*
* Step 0:
*
      N = 2**log_2_N
      halfN = N/2

      DO i = 1, pixels
        DO j = 1, pixels
          image(i,j) = 0.d0
        END DO
      END DO
      ...
*
* Loop over Steps 1-5:
*
      DO count = 1, num_phi
        phi = ...
*
* STEP 1:
*
        < get the total projection at phi >
*
* STEP 2:
*
        call fft( temporary, log_2_N, 0)

```

```

*
* STEP 3:
*
*       < multiply TEMPORARY by absolute value of k >
*       call fft( temporary, log_2_N, 1)
*
* STEP 4:
*
*       < update IMAGE ... >
*
* STEP 5:
*
*       < display IMAGE ... >
*
* STEP 6:
*
*       END DO
*       end

```

Let's look at this in more detail. To obtain the total projection at ϕ , we need to make repeated calls to `CT_PROJECTION(xi, phi)`. Let's imagine that we want to sample the projection at increments of $\Delta\xi = 0.1$, starting at $\xi = -1.0$, giving us 21 sampled points. We'll add another 11 points, bringing the total to 32, a number appropriate for use with the FFT, and hence start the sampling at $\xi = -1.6$. It's important to note that these additional points are not "wasted" in any sense — it's just as important to know where the projection is zero, as to know where it isn't! With these considerations, Step 1 might look something like this:

```

*
* STEP 1:
*
*       delta_xi = 0.1d0
*       DO i = 1, N
*           xi(i) = -1.6d0 + delta_xi * dble(i-1)
*           projection(i) = ct_projection(xi(i), phi)
*       END DO

```

We then move on to Step 2, Fourier-transforming the projection. There are two crucial considerations to be made here, the first being the number of points to be used in the Fourier transform. We have already decided to use 32 points in the transform, but this should be regarded as a provisional decision. While we have adequately sampled the *range* of the projection, it is possible that we haven't sampled often enough. And of course, we also need

to consider the range and resolution of k_ξ in transform space.

The second consideration regards the manner of storage. The FFT expects — no, *requires!* — the data to be in a specific order: the first entry corresponds to the coordinate's being zero, e.g., $\xi = 0$, and increases. But the data are also periodic, so that the last entry in the array corresponds to $\xi = -\Delta\xi$. This is not the order in which the projection data are obtained (and stored), however. There are several remedies for this data mismatch, the easiest of which is to simply reorder the projection data. That is, the data are shuffled into a different array to be used in the FFT subroutine. Then we can perform Step 2, Fourier-transforming the data stored in TEMPORARY.

```
*
* STEP 2
*
      DO i = 1, halfN
        temporary( i      ) = projection(i + halfN)
        temporary(i + halfN) = projection( i      )
      END DO
      call fft( temporary, log_2_N, 0)
```

Step 3 is straightforward, if we remember that the data are stored in transform space in the same periodic manner as they are in coordinate space. Thus multiplication by $|k_\xi|$ is accomplished by

```
*
* STEP 3
*
      delta_k = 2.d0 * pi / (N * delta_x)
      DO i = 1, halfN
        k              = dble(i-1) * delta_k
        temporary(i)   = abs(k) * temporary(i)
        k              = -dble( i ) * delta_k
        temporary(N+1-i) = abs(k) * temporary(N+1-i)
      END DO
```

The inverse transform then yields the modified projection, in the wraparound storage scheme. It will be helpful to return the data to their original ordering,

```
      DO i = 1, halfN
        projection( i      ) = REAL( temporary(i+halfN) )
        projection(i+halfN) = REAL( temporary( i      ) )
      END DO
```

We can now evaluate the contribution from this projection to the image, i.e., to all the pixels in the xy -image plane. Cycling over all the 51×51 elements of the image, Step 4 might look like this:

```

*
* STEP 4
*
      DO i = 1, pixels
        x = -1.d0 + dble(i-1)*2.d0 / dble(pixels-1)
        DO j = 1, pixels
          y = -1.d0 + dble(j-1)*2.d0 / dble(pixels-1)
          xi_value = x * cos(phi) + y * sin(phi)
*
*      Find "index" such that xi(index) < xi < xi(index+1)
*
          index = (xi_value + 1.6d0)/delta_xi + 1.d0
*
*      Interpolate
*
          image(i,j) = image(i,j)
+          + ((xi_value - xi(index))*projection(index+1) +
+            (xi(index+1)-xi_value)*projection( index ) )
+          /delta_xi
          END DO
        END DO

```

It's not necessary to display the image after every additional projection. However, it's interesting to see the image build in this way and so let's do it at least this once.

```

*
* STEP 5
*
      DO i = 1, pixels
        DO j = 1, pixels
          < index = ???, depending upon image(i,j) >
          call color( index )
*
* Remember, Subroutine PIXEL counts from top to bottom!
*
          call pixel( i+100, 200-j )
        END DO
      END DO

```

EXERCISE 6.18

Using the outline provided, write a program to reconstruct the test image.

Depending upon the computer you are using, the reconstructed image might appear rather small. After all, 51×51 pixels is not very large area.

EXERCISE 6.19

“Magnify” the image by increasing the number of points used in the image to, say, 101×101 . Note that this modification is independent of both the number of points in a single projection and the number of projections.

While this last exercise yields a larger image, little (any?) of the “fuzziness” that was present in the original reconstruction has been eliminated. Simply by virtue of its increased size, more detail is visible — but is the detail real, or an artifact of the reconstruction? (Since we know what the image is supposed to be, Figure 6.23, we in fact know that it’s an artifact.) We can try to improve the image by replacing the simple linear interpolation with something more sophisticated. Perhaps cubic interpolation would help...

EXERCISE 6.20

Modify the reconstruction program by using cubic interpolation to obtain the specific value of the modified projection at ξ .

Regardless of the results of the last exercise, even more improvement can be obtained. There are at least three further variables at your disposal for improving the quality of the image: the number of points in each projection; the range of ξ in each projection; and the number of projections. Recall that $\Delta\xi$ is inversely related to the range of k_ξ , so that you might want to vary the range of ξ and the number of points in a projection simultaneously. These variables are somewhat independent of the number of projections — you can always add more projections until the image quality stabilizes.

EXERCISE 6.21

Investigate improvements in the image quality by modifying these variables.

The real test of your understanding of computerized tomography is in constructing an image of an unknown object, of course. In the FCCP library

is a subroutine named `Unknown_Projection` having the same arguments and serving in the same role as `CT_Projection`, but of an unknown object.

EXERCISE 6.22

Construct an image of the UNKNOWN object.

Astrophysicists face a problem remarkably similar to that posed by computerized tomography when they attempt to create images from the interference patterns obtained by radiotelescopes. There are additional complexities, but the basic similarity exists: the signal that is measured is related via Fourier transforms to the source of the signals. Radar imaging, either terrestrial or planetary, also involves the manipulation of data and its Fourier transforms. These applications are computationally intensive and require considerable computer power. But as more powerful computers become available we can expect such applications to increase both in number and in complexity.

References

Fourier analysis is an extremely valuable tool in physics, applied physics, and engineering. As a result, it's discussed in many texts on the subject, such as

George Arfken, *Mathematical Methods for Physicists*, Academic Press, New York, 1985.

Mary L. Boas, *Mathematical Methods in the Physical Sciences*, John Wiley & Sons, New York, 1983.

Sadri Hassani, *Foundations of Mathematical Physics*, Allyn and Bacon, Boston, 1991.

There are also a wide variety of excellent references devoted exclusively to Fourier analysis. One of the very best for physical scientists is

David C. Champeney, *Fourier Transforms and Their Physical Applications*, Academic Press, New York, 1973.

There's also considerable literature on the FFT, such as

E. Oran Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

And there are applications of Fourier analysis to specific areas of physics, such as

E.G. Steward, *Fourier Optics: An Introduction*, John Wiley & Sons, New York, 1987.

References to radar are also quite extensive. An excellent place to begin is

M. I. Skolnik, *Introduction to Radar Systems*, McGraw-Hill, New York, 1980.

The topic of computerized tomography can be quite complicated and is not usually presented at an introductory level. If you want to study it more, you might consider the following:

R. H. T. Bates and M. J. McDonnell, *Image Restoration and Reconstruction*, Claredon Press, Oxford, 1986.

F. Natterer, *The Mathematics of Computerized Tomography*, John Wiley & Sons, New York, 1986.

Image Reconstruction from Projections, edited by Gabor T. Herman, Springer-Verlag, Berlin, 1979.

Chapter 7:

Partial Differential Equations

As we consider the “more advanced” topics of physics, such as advanced classical mechanics, electromagnetic theory, and quantum mechanics, we find a profusion of *partial differential equations* used to describe the physical phenomena. There are times when these equations can be simplified, or perhaps special cases considered, so that the problem is reduced to one involving ordinary differential equations, which are generally easier to solve. More generally, however, a numerical approach is called for.

Earlier, we saw that derivatives can be approximated by differences, and differential equations thus transformed into difference equations. In this chapter the finite difference method, introduced in Chapter 5, will be extended to the multidimensional case appropriate for partial differential equations. We’ll also investigate a radically different method of solution, one that utilizes the FFT with dramatic efficiency for certain problems.

Classes of Partial Differential Equations

As noted above, many of the interesting equations of physics are partial differential equations. That is, the physical problem involves two or more independent variables, such as an x -coordinate and time, and is described as having specific relationships existing between various derivatives. For example, we have the *wave equation*

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \quad (7.1)$$

in one spatial dimension, where c is the velocity of the wave. The equation simply states that the second derivatives of the “solution” are proportional to one another, with proportionality constant c^2 .

Other equations express different relationships between the derivatives. For example, if only the first derivative with respect to time appears, we have the *diffusion equation*,

$$\frac{\partial u}{\partial t} - k \frac{\partial^2 u}{\partial x^2} = 0. \quad (7.2)$$

This is also known as the *heat equation*, as it describes the flow (diffusion) of heat through a conductor. (In this case, k is the thermal conductivity — its reciprocal, R , is the thermal resistance, e.g., the insulation quality of a material.) If an object is isolated from the environment, eventually it will reach a steady-state distribution of temperature, an equilibrium condition in which the time derivative is zero. This is the *Laplace equation*,

$$\frac{\partial^2 u}{\partial x^2} = 0 \quad (7.3)$$

in one dimension. More interesting is the three-dimensional case, for which Laplace's equation becomes

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0, \quad (7.4)$$

where ∇^2 is called the *Laplacian*. Laplace's equation states that the Laplacian is zero everywhere — *Poisson's equation* states that the Laplacian exhibit a specific spatial dependence,

$$\nabla^2 u - f(x, y, z) = 0. \quad (7.5)$$

This allows for the introduction of heat “sources” and “sinks” into the problem. Like Laplace's equation, the solutions to Poisson's equation are independent of time.

Another equation of common interest, not too unlike Poisson's equation, is the *Helmholtz equation*,

$$\nabla^2 u + \lambda u = 0. \quad (7.6)$$

Last in our brief list, but certainly not least in importance, is the *Schrödinger equation*,

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(x, y, z) \right] u - i\hbar \frac{\partial u}{\partial t} = 0. \quad (7.7)$$

(We note that the Schrödinger equation is also a diffusion equation, but its fundamental importance to quantum mechanics warrants its special mention.)

Let's consider a general (linear) second-order partial differential equation in two variables, say, x and t , such as

$$A \frac{\partial^2 u}{\partial x^2} + B \frac{\partial^2 u}{\partial x \partial t} + C \frac{\partial^2 u}{\partial t^2} + D(x, t, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial t}) = 0, \quad (7.8)$$

where A , B , and C are functions of x and t and D is a function of u and the derivatives $\partial u / \partial x$ and $\partial u / \partial t$ as well as x and t . Now, we might try to introduce new variables such that the terms involving mixed derivatives vanish. In so doing, we would find that the discriminant $B^2 - 4AC$ plays an important role. This same quantity appears in the analysis of conic sections — that is, the shape of the curve resulting from the intersection of a plane with a cone. In that case, there are three distinct possibilities: the hyperbola, the parabola, and the ellipse. By analogy, we employ the same terminology here. That is, if

$$B^2(x_0, t_0) - 4 A(x_0, t_0) C(x_0, t_0) > 0 \quad (7.9)$$

at (x_0, t_0) , then the partial differential equation is said to be *hyperbolic at the point* (x_0, t_0) . If the equation is hyperbolic at all points in the domain of interest, then it is said to be a *hyperbolic equation*. For example, since the velocity c is a real number, the wave equation, Equation (7.1), is a hyperbolic equation. If

$$B^2(x_0, t_0) - 4 A(x_0, t_0) C(x_0, t_0) = 0, \quad (7.10)$$

the equation is *parabolic at the point* (x_0, t_0) . The diffusion equation is an example of a parabolic partial differential equation. And finally, if

$$B^2(x_0, t_0) - 4 A(x_0, t_0) C(x_0, t_0) < 0, \quad (7.11)$$

the equation is *elliptic at the point* (x_0, t_0) — Laplace's and Poisson's equations are examples of elliptic equations.

The Vibrating String ... Again!

As our first example of a physical problem described in terms of partial differential equations, we'll investigate the motion of waves on a string. We'll begin with an *ideal* string, one that's perfectly elastic, offers no resistance to bending, and is stretched between two immovable supports. We'll further assume that the mass density of the string is uniform, that the tension in the

string is far greater than the weight of the string — so that we can ignore the effect of gravity — and that the amplitude of the string's motion is very small. We thus have a naive model of a violin string, for example.

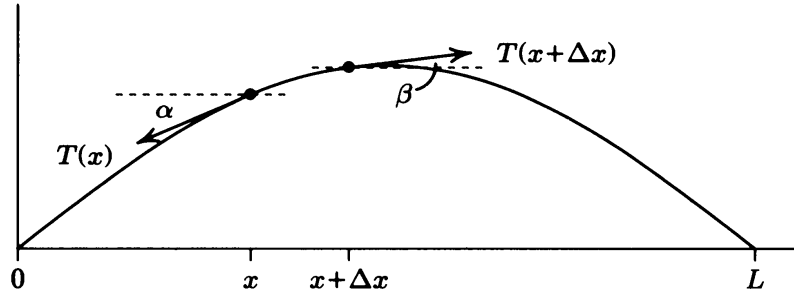


FIGURE 7.1 A vibrating string.

We'll consider a small element of that string, as in Figure 7.1, and apply Newton's second law to it. There are two forces acting on the element, the tension at x and at $x + \Delta x$. The vertical component of the net force is simply

$$F_y = -T(x) \sin \alpha + T(x + \Delta x) \sin \beta. \quad (7.12)$$

If the displacement of the string is small, the angles α and β are small, so that the *sine* of an angle can be replaced by the *tangent* of the angle,

$$\sin \alpha \approx \tan \alpha \quad \text{and} \quad \sin \beta \approx \tan \beta. \quad (7.13)$$

But, of course, the tangent of the angle is just the slope of the curve, the derivative $\partial u / \partial x$, so that

$$F_y = -T(x) \left. \frac{\partial u}{\partial x} \right|_x + T(x + \Delta x) \left. \frac{\partial u}{\partial x} \right|_{x+\Delta x}. \quad (7.14)$$

For our problem, the tension in the string is the same everywhere, so that $T(x) = T(x + \Delta x)$. Expressing the derivative at $x + \Delta x$ in a Taylor series about x , we have

$$\begin{aligned} F_y &= -T \left. \frac{\partial u}{\partial x} \right|_x + T \left[\left. \frac{\partial u}{\partial x} \right|_x + \Delta x \frac{\partial}{\partial x} \left. \frac{\partial u}{\partial x} \right|_x + \cdots \right] \\ &\approx T \Delta x \frac{\partial^2 u}{\partial x^2}. \end{aligned} \quad (7.15)$$

Of course, Newton's second law tells us that the net force produces an acceleration, $\mathbf{F} = m\mathbf{a}$. If μ is the mass density of the string, then $\mu \Delta x$ is the mass

of an element of the string and

$$\mu \Delta x \frac{\partial^2 u}{\partial t^2} = T \Delta x \frac{\partial^2 u}{\partial x^2}. \quad (7.16)$$

Canceling the length element, we arrive at the wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{T}{\mu} \frac{\partial^2 u}{\partial x^2}. \quad (7.17)$$

Comparing to Equation (7.1), we conclude that the velocity of a wave on the string is

$$c = \sqrt{\frac{T}{\mu}}. \quad (7.18)$$

Finite Difference Equations

We'll devise a numerical approach to the solution of the wave equation using finite difference approximations to the derivatives. We've already used finite differences for problems involving one independent variable — it works the same with two (or more). Recall that the second derivative can be approximated by

$$\frac{d^2 f(x)}{dx^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2). \quad (7.19)$$

In two dimensions, we impose a rectangular grid in space and time such that

$$\begin{aligned} x_i &= ih_x, & i &= 0, 1, \dots, N_x, \\ t_j &= j\delta_t, & j &= 0, 1, \dots, N_t. \end{aligned} \quad (7.20)$$

The finite difference approximation to the wave equation is then

$$\frac{u_i^{j+1} - 2u_i^j + u_i^{j-1}}{\delta_t^2} - c^2 \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h_x^2} = 0, \quad (7.21)$$

where we've introduced the notation

$$u_i^j = u(x_i, t_j). \quad (7.22)$$

Solving for u_i^{j+1} , we find

$$u_i^{j+1} = \frac{\delta^2 c^2}{h^2} (u_{i+1}^j + u_{i-1}^j) + 2(1 - \frac{\delta^2 c^2}{h^2}) u_i^j - u_i^{j-1}. \quad (7.23)$$

This equation tells us that if we know u at all x_i at the times t_j and t_{j-1} , then we can immediately determine u at all x_i at the next time step, t_{j+1} . It is said to be an *explicit* method for determining the solutions.

There is a small difficulty in getting started, however, since we usually won't know u at two successive times. Rather, we might know $u(x_i, 0)$ and the derivative $\partial u(x_i, 0)/\partial t$ at all x_i . Then we have

$$\left. \frac{\partial u(x_i, t)}{\partial t} \right|_{t=0} = \frac{u_i^1 - u_i^{-1}}{2\delta} \quad (7.24)$$

or

$$u_i^{-1} = u_i^1 - 2\delta \left. \frac{\partial u(x_i, t)}{\partial t} \right|_{t=0}. \quad (7.25)$$

With this expression for u_i^{-1} , we can write u_i^1 as

$$u_i^1 = \frac{\delta^2 c^2}{2h^2} (u_{i+1}^0 + u_{i-1}^0) + (1 - \frac{\delta^2 c^2}{h^2}) u_i^0 + \delta \frac{\partial u(x_i, 0)}{\partial t}. \quad (7.26)$$

These equations are readily transformed into a suitable computer code. Perhaps the most straightforward way to proceed is to introduce one array to hold u at all x_i at the time t_j , a second array to hold all the u at the time t_{j-1} , and a third to hold the newly computed results at t_{j+1} . Then we loop through the code, incrementing the time and shuffling the arrays appropriately. An outline of the subroutine might look like the following:

```

Subroutine string
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* STRING investigates the motion of a string by solving the
* time dependent wave equation by finite differences.
*
      double precision Uold(50), U(50), Unew(50)
      double precision U_0(50), dU_0(50)
      double precision time, dt, dx, epsilon, T, mu, c
      integer i, Nx, Nt
*
*   U_0 is the initial u at t=0, and dU_0 is the time

```

```

*   derivative evaluated at t=0.  These are specified as
*   initial conditions of the problem.
*   These arrays must be initialized.  This can be done
*   through a parameter statement, or by evaluating
*   appropriate functions inside a DO-loop.
*
*   ...
*
*       N = number of points in the x-grid
*       dx = space increment
*       dt = time increment
*       T = tension
*       mu = mass density
*       c = wave velocity
*
*   epsilon = (dt*c/dx)**2
*
*   Initialize U and Uold.      Let Uold = u(0), U = u(1 dt).
*
*       time = 0.d0
*       DO i = 1, n
*           Uold(i) = U_0(i)
*       END DO
*
*   U(endpoints) are fixed, = 0.
*
*       U(1) = 0.d0
*       U(n) = 0.d0
*       DO i = 2, n-1
*           U(i) = 0.5d0 * epsilon * ( U_0(i+1) + U_0(i-1) ) +
+               (1.d0-epsilon) * U_0(i) + dt * dU_0(i)
*       END DO
*
*   Start main loop
*
1000   time = time + dt
        Unew(1) = 0.d0
        Unew(n) = 0.d0
        DO i = 2, n-1
            Unew(i) = epsilon * ( U(i+1) + U(i-1) ) +
+                2.d0 * (1.d0-epsilon) * U(i) - Uold(i)
        END DO
*

```

```

*  Shuffle arrays ...
*
      DO i = 1, n
        Uold(i) = U(i)
        U(i)    = Unew(i)
      END DO
      IF ( time .lt. MaxTime) goto 1000
end

```

This subroutine can be used to “solve” the problem — that is, to generate a large amount of data over many time steps. But data in such a form is nearly impossible to comprehend. On the other hand, a visual representation of the data is readily understood. If we could display the solutions at successive time steps, a primitive level of animation would be achieved. In the present case, such a display would vividly display the motion of the wave. And in fact, it’s not that difficult to do! As we plot the solution at the current time, we simply erase the previous results from the display. An appropriate subroutine for this task is then

```

      Subroutine Animate( x, old, new, n , OFF, ON )
      Double precision x(1), old(1), New(1)
      Integer i, n, OFF, ON
*
*  This subroutine "erases" the data in OLD and then
*  plots the data in NEW. To provide the most aesthetic
*  animation, the old-erasing and new-plotting is done
*  one line segment at a time.
*
*      OFF = the background (erasing) color index
*      ON  = the color index used to draw the line
*
      DO i = 2, n
        call color ( OFF )
        call line( x(i-1), old(i-1), x(i), old(i) )
        call color ( ON  )
        call line( x(i-1), new(i-1), x(i), new(i) )
      END DO
end

```

The computer code we’ve developed can be applied to a variety of problems. Let’s start with a meter length of string, stretched with a tension of 10 Newtons and having a mass of 1 gram. Initially, we’ll deform the string so that it

has a “bump” in the middle,

$$u(x, 0) = \begin{cases} 0, & x = 0, \\ e^{-100(x-0.5)^2}, & 0 < x < 1, \\ 0, & x = 1. \end{cases} \quad (7.27)$$

and is motionless at time $t = 0$. Take $h = 1$ cm, and $\delta_t = 0.0001$ seconds.

EXERCISE 7.1

Use the computer code outlined above, with the parameters discussed, to calculate and observe the motion of the string for 20 time steps.

You probably observed a rather curious result — the bump “fell apart” into two bumps, moving in opposite directions. This is a very general result. Solutions to the wave equation are *waves* — depending upon the initial conditions, the general solution to the equation is a combination of one waveform moving to the right and a second waveform moving to the left. If, as we required, the initial time derivative is zero and the initial waveform is symmetric, then the waveform has no choice but to evolve as it did. To demonstrate, write

$$u(x, t) = f(x + ct) + g(x - ct), \quad (7.28)$$

where f and g are the waveforms moving to the left and right, respectively. Then

$$\begin{aligned} \frac{\partial^2 u(x, t)}{\partial t^2} &= \frac{\partial}{\partial t} \left[\frac{\partial}{\partial t} (f(x + ct) + g(x - ct)) \right] \\ &= \frac{\partial}{\partial t} \left[c \frac{df(x + ct)}{dt} - c \frac{dg(x - ct)}{dt} \right] \\ &= c^2 \frac{d^2 f}{dt^2} + c^2 \frac{d^2 g}{dt^2} \end{aligned} \quad (7.29)$$

and

$$\begin{aligned} \frac{\partial^2 u(x, t)}{\partial x^2} &= \frac{\partial}{\partial x} \left[\frac{\partial}{\partial x} (f(x + ct) + g(x - ct)) \right] \\ &= \frac{\partial}{\partial x} \left[\frac{df(x + ct)}{dx} + \frac{dg(x - ct)}{dx} \right] \\ &= \frac{d^2 f}{dx^2} + \frac{d^2 g}{dx^2}. \end{aligned} \quad (7.30)$$

Clearly then, our guess of Equation (7.28) is a solution to the wave equation of

Equation (7.1), and we've shown that the general solution of the wave equation is a linear combination of two counter-propagating waves.

Perhaps just as curious, if not more so, is *how well* the numerical method worked in the exercise. After all, the derivatives were only second-order accurate, yet the solutions appear very good, much better than we might have expected. Indeed, we have stumbled upon a very fortuitous combination which gives rise to *exact* results! For our particular choice of h_x and δ_t , in conjunction with the mass density and tension of the string, we have

$$c = \sqrt{\frac{T}{\mu}} = 100 \text{ m/sec} \quad (7.31)$$

so that

$$\frac{\delta_t^2 c^2}{h_x^2} = 1 \quad (7.32)$$

or

$$h_x = c\delta_t. \quad (7.33)$$

For this particular choice, Equations (7.23) and (7.26) reduce to

$$u_i^{j+1} = u_{i+1}^j + u_{i-1}^j - u_i^{j-1} \quad (7.34)$$

and

$$u_i^1 = \frac{u_{i+1}^0 + u_{i-1}^0}{2} - \delta_t \frac{\partial u(x_i, 0)}{\partial t}. \quad (7.35)$$

If we thought of the two-dimensional grid painted alternately black and red, as on a chess board, we see that the values of the red squares are determined solely from other red squares, and black squares are determined from other black squares. The calculational grid has decomposed itself into two, noninteracting grids: the reds and the blacks. But, of course, this doesn't explain why the calculation is so accurate.

Using Taylor's series, we can write

$$u_{i+1}^j = u_i^j + h_x \frac{\partial u_i^j}{\partial x} + \frac{h_x^2}{2!} \frac{\partial^2 u_i^j}{\partial x^2} + \frac{h_x^3}{3!} \frac{\partial^3 u_i^j}{\partial x^3} + \frac{h_x^4}{4!} \frac{\partial^4 u_i^j}{\partial x^4} + \dots \quad (7.36)$$

and

$$u_{i-1}^j = u_i^j - h_x \frac{\partial u_i^j}{\partial x} + \frac{h_x^2}{2!} \frac{\partial^2 u_i^j}{\partial x^2} - \frac{h_x^3}{3!} \frac{\partial^3 u_i^j}{\partial x^3} + \frac{h_x^4}{4!} \frac{\partial^4 u_i^j}{\partial x^4} - \dots \quad (7.37)$$

Adding these equations and rearranging terms, we easily find

$$\frac{\partial^2 u_i^j}{\partial x^2} = \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h_x^2} - 2\frac{h_x^2}{4!} \frac{\partial^4 u_i^j}{\partial x^4} - 2\frac{h_x^4}{6!} \frac{\partial^6 u_i^j}{\partial x^6} + \dots \quad (7.38)$$

In like fashion, we find

$$\frac{\partial^2 u_i^j}{\partial t^2} = \frac{u_i^{j+1} - 2u_i^j + u_i^{j-1}}{\delta_t^2} - 2\frac{\delta_t^2}{4!} \frac{\partial^4 u_i^j}{\partial t^4} - 2\frac{\delta_t^4}{6!} \frac{\partial^6 u_i^j}{\partial t^6} + \dots \quad (7.39)$$

Substituting these expressions into the wave equation, we find

$$\begin{aligned} \frac{\partial^2 u_i^j}{\partial t^2} - c^2 \frac{\partial^2 u_i^j}{\partial x^2} &= \frac{u_i^{j+1} - 2u_i^j + u_i^{j-1}}{\delta_t^2} - c^2 \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h_x^2} \\ &\quad - 2 \left\{ \frac{\delta_t^2}{4!} \frac{\partial^4 u_i^j}{\partial t^4} - c^2 \frac{h_x^2}{4!} \frac{\partial^4 u_i^j}{\partial x^4} + \frac{\delta_t^4}{6!} \frac{\partial^6 u_i^j}{\partial t^6} - c^2 \frac{h_x^4}{6!} \frac{\partial^6 u_i^j}{\partial x^6} + \dots \right\}. \end{aligned} \quad (7.40)$$

That is, if the terms in the curly brackets are zero, the finite difference expression is exactly equal to the exact wave equation. But consider — for exact solutions, we have

$$\frac{\partial^4 u}{\partial t^4} = \frac{\partial^2}{\partial t^2} \left[\frac{\partial^2 u}{\partial t^2} \right] = \frac{\partial^2}{\partial t^2} \left[c^2 \frac{\partial^2 u}{\partial x^2} \right] = c^2 \frac{\partial^2}{\partial x^2} \left[\frac{\partial^2 u}{\partial t^2} \right] = c^2 \frac{\partial^2}{\partial x^2} \left[c^2 \frac{\partial^2 u}{\partial x^2} \right] = c^4 \frac{\partial^4 u}{\partial x^4}, \quad (7.41)$$

so that

$$\frac{\delta_t^2}{4!} \frac{\partial^4 u}{\partial t^4} - c^2 \frac{h_x^2}{4!} \frac{\partial^4 u}{\partial x^4} = \frac{1}{4!} (\delta_t^2 c^4 - h_x^2 c^2) \frac{\partial^4 u}{\partial x^4}. \quad (7.42)$$

This is the leading term in the error. But for the very specific choice that $h_x = c\delta_t$, the term vanishes! In fact, one can easily find that all the terms vanish, so that the finite difference solution is the exact solution!

(Actually, we need to qualify this statement a bit. We have just determined the error in using Equation (7.23); but we also required Equation (7.26) to begin our solution. If this were a poor approximation to u_i^1 , then our calculated solution would suffer.)

We thus find a rather unusual circumstance — if we decrease the time step, thinking that we will improve the accuracy of the calculation, we will actually *increase* the error! This is a consequence of the very unique nature of this particular equation, and certainly is not true for partial differential equations in general.

Let's see if we can construct a wave that is moving to the left. To do this, we need a function that depends upon the particular combination $x + ct$, such as

$$u(x, t) = \begin{cases} 0, & x = 0, \\ e^{-100(x+ct-0.5)^2}, & 0 < x < 1, \\ 0, & x = 1. \end{cases} \quad (7.43)$$

The time derivative of this function is

$$\left. \frac{\partial u}{\partial t} \right|_{t=0} = -200c(x - 0.5)e^{-100(x+ct-0.5)^2}, \quad 0 < x < 1, \quad (7.44)$$

and zero at the rigid endpoints.

EXERCISE 7.2

Verify that this function represents a wave moving to the left, by following the time evolution for 20 time steps.

We can, of course, easily observe the phenomena of constructive and destructive interference of waves. Let the displacement of the string be written as

$$u(x, t) = \alpha f(x, t) + \beta g(x, t) \quad (7.45)$$

where $f(x, t)$ and $g(x, t)$ are two waves moving on the string. We'll choose $f(x, t)$ to be initially centered at $x = 0.75$ and moving to the left,

$$f(x, 0) = e^{-100(x-0.75)^2}, \quad (7.46)$$

with derivative

$$\left. \frac{\partial f}{\partial t} \right|_{t=0} = -200c(x - 0.75)e^{-100(x-0.75)^2}, \quad 0 < x < 1, \quad (7.47)$$

and $g(x, t)$ to be initially centered at $x = 0.25$ and moving to the right,

$$g(x, 0) = e^{-100(x-0.25)^2}, \quad (7.48)$$

with derivative

$$\left. \frac{\partial g}{\partial t} \right|_{t=0} = +200c(x - 0.25)e^{-100(x-0.25)^2}, \quad 0 < x < 1. \quad (7.49)$$

EXERCISE 7.3

Show that constructive interference occurs when α and β have the same sign, while the waves interfere destructively if they have opposite signs.

Thus far, the wave has been kept away from the ends, where it is rigidly supported. But the interaction of the wave with the supports is very important — in particular, the support will *reflect* the wave.

EXERCISE 7.4

Follow the wave disturbance for a longer time, say, 500 time steps.

You observed, of course, the phenomena of *phase reversal* as the wave was reflected from the rigid support. What happens if the support is not rigid?

Let's imagine that the string is free at $x = 0$. The vertical component of the force on the end of the string is

$$F_y = -T(0) \sin \alpha \approx -T(0) \left. \frac{\partial u}{\partial x} \right|_{x=0}. \quad (7.50)$$

But there's no physical support to provide the force, so the force must be zero! Hence

$$\left. \frac{\partial u}{\partial x} \right|_{x=0} = 0. \quad (7.51)$$

In terms of a difference equation, we have

$$\frac{u_1^j - u_0^j}{h} = 0, \quad (7.52)$$

or simply

$$u_0^j = u_1^j, \quad \text{for all } j. \quad (7.53)$$

To treat a free end we simply replace our fixed boundary condition, that $u_0^j = 0$, by this one.

EXERCISE 7.5

Modify your program to treat the string as free at $x = 0$, and repeat the previous exercise.

That the supports of real strings move is not a trivial observation. In fact, virtually all stringed musical instruments use this feature to generate

sound. With a violin, for example, the sound doesn't emanate from the vibrating string — it comes from the vibrating violin, set in motion by the string. That is, the vibration of the string is coupled to the vibration of the violin box to generate the sound. If the string supports were truly rigid, the box would not vibrate, and the sound we recognize as the violin would not be generated.

The supports for real strings, such as found on musical instruments, lie somewhere between being perfectly rigid and being free. One way we can at least begin to consider real strings is by modeling the support as having mass: the free end corresponds to zero mass, and the rigid end corresponds to infinite mass. Real supports lie between these extremes.

We've already calculated the force on the string, Equation (7.50). We now apply Newton's second law to the support to find

$$M \frac{\partial^2 u}{\partial t^2} \Big|_{x=0} = T \frac{\partial u}{\partial x} \Big|_{x=0}, \quad (7.54)$$

where M is the effective mass of the support. Our finite difference expression for the condition at the boundary is then

$$u_0^{j+1} = \left(2 - \frac{T\delta^2}{Mh}\right) u_0^j - u_0^{j-1} + \frac{T\delta^2}{Mh} u_1^j. \quad (7.55)$$

EXERCISE 7.6

Modify your program appropriately. Compare the behavior of the system with $M = 0.001$ kg to that when the end was fixed and when it was free.

We could continue to make our description of supports, and hence our boundary conditions, more realistic. For example, we could model the support as being a mass on a spring, damped by the loss of energy to the support (and the soundboard). Then the force might be modeled as

$$M \frac{\partial^2 u}{\partial t^2} \Big|_{x=0} + R \frac{\partial u}{\partial t} \Big|_{x=0} + Ku(0, t) = T \frac{\partial u}{\partial x} \Big|_{x=0}, \quad (7.56)$$

where K is a spring constant, or stiffness, and R is a damping coefficient. Real musical instruments, such as a violin, are further complicated by the presence of a large number of vibrational modes in the instrument itself. These modes couple to one another, and to the string, providing a richness that is both complicated to describe mathematically, and pleasing to the ear.

The Steady-State Heat Equation

In Chapter 5 we used the finite difference method to solve a two-point boundary value problem in one dimension. However, the real importance of these methods lies not in one dimension, but in their application to multidimensional problems, e.g., partial differential equations. For example, the heat flow in a homogeneous body is governed by the heat equation

$$\nabla^2 u(x, y, z, t) = \frac{1}{c^2} \frac{\partial u(x, y, z, t)}{\partial t}, \quad c^2 = \frac{K}{\sigma \rho}, \quad (7.57)$$

where u is the temperature, K is the thermal conductivity, σ is the specific heat, and ρ is the density. If we consider a thin metal plate, the problem reduces to two dimensions, and the steady-state distribution of temperature in the plate will then satisfy the two-dimensional Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0. \quad (7.58)$$

Of course, we must know something of the boundary conditions before we can solve the problem. Suppose, for example, that we know the temperature at all points on the boundary of the plate. This is a common situation, known as *Dirichlet* boundary conditions, and it allows us to determine a unique solution to the problem.

After having solved the two-point boundary value problem in one dimension, the plan of attack for this two-dimensional problem is rather obvious: we replace the partial derivatives by their finite difference approximation, and solve the resulting finite difference equations by the method of successive over-relaxation. Recall that the second derivative can be approximated by

$$\frac{d^2 f(x)}{dx^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2). \quad (7.59)$$

For a rectangular plate of dimension $a \times b$ we introduce a grid such that

$$\begin{aligned} x_i &= ih_x, \quad i = 0, 1, \dots, N_x, \\ y_j &= jh_y, \quad j = 0, 1, \dots, N_y. \end{aligned} \quad (7.60)$$

We then easily find the finite difference equation

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} = 0, \quad (7.61)$$

where we've introduced the notation $u_{i,j} = u(x_i, y_j)$. Solving for $u_{i,j}$, we find

$$u_{i,j} = \frac{h_x^2 h_y^2}{2h_x^2 + 2h_y^2} \left[\frac{u_{i+1,j} + u_{i-1,j}}{h_x^2} + \frac{u_{i,j+1} + u_{i,j-1}}{h_y^2} \right]. \quad (7.62)$$

Of course, if $h_x = h_y$ the expression takes on the much simpler form

$$u_{i,j} = \frac{1}{4} [u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}]. \quad (7.63)$$

The manner of solution is exactly as for the two-point boundary value problem, except that we now loop over both x - and y -coordinates. For example, the code to solve the heat equation on a 9×9 grid might be patterned after the following:

```

Subroutine HEAT2D
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine implements Successive Over-Relaxation
* to solve the 2D Heat equation on a 9 x 9 grid.
*
      Double Precision u(9,9), hx,hy,...
      Integer count, i, j, ...
      logical done
      ...
      hx = ...
      hy = ...
      alpha = ...
      CALL CLEAR
*
* Initialize u(i,j) --- to help keep things straight,
*           i will always refer to the x-coordinate,
*           j will always refer to the y-coordinate.
*
      DO i = 1, 9
        DO j = 1, 9
          u(i,j) = ...
        END DO
      END DO
*
* Initialize values on the boundaries
*
      DO ...

```

```

        u(1,j) = ...
        ...
    END DO
    count = 0

*
* Iteration starts here...
*
100    DONE = .TRUE.                !    <-----    LOOK    HERE
        count = count + 1
        IF ( count .GE. 100 ) stop 'Too many iterations!'

*
* The following DO loops do the actual SOR, using the
* relaxation parameter alpha:
*
        DO i = 2, 8
            DO j = 2, 8
*
*                                     "u(i,j)" is the old value,
*                                     "uu" is the latest iterate.
*
                ...
                uu =
                IF(dabs((uu-u(i,j))/uu).gt.5.d-5)DONE = .FALSE.
*                Set u(i,j) = over relaxed value
                u(i,j) = uu + alpha * (uu-u(i,j))
            END DO
        END DO

        call cursor(1,1)
        write(*,1000)
        write(*,1000)

*
* Note the order in which the function is written to
* the screen. On the screen, u(1,1) will be on the
* lower left side; i increases to the right, j increases
* upwards.
*
        DO j = 1, 9
            write(*,1000)(u(i,10-j),i=1,9)
        END DO
1000    format(2x,9f8.3)
        write(*,2000)count
2000    format(/,25x,'Iteration = ',i3)

        IF (.NOT.DONE)goto 100
    end

```

The functioning of this code fragment should be pretty clear. Various quantities are initialized before the iteration loop, as well as the function itself. Note that *every* value of the function should be initialized, not just the ones on the boundary.

The functional part of this two-dimensional code is quite similar to the previous one-dimensional example. We have introduced a new subroutine, CLEAR:

CLEAR ... clears the screen, in either graphics or text mode.

In combination with CURSOR, the effect is to clear the screen and return the cursor to the upper left corner of the screen. Subsequent WRITE statements then overwrite whatever is present on the screen. Calling these routines in the order indicated gives the effect of updating the screen, and hence the function values, without totally erasing and rewriting the screen.

We previously indicated that there exists an optimum α for every situation, dictated by the structure of the problem. As discussed in several numerical analysis texts, the optimum value can be derived analytically in certain cases and can be estimated in others. Using the second-order central difference approximations in two Cartesian coordinates, the optimal value for the relaxation parameter is given as

$$\alpha = \frac{4}{2 + \sqrt{4 - [\cos \frac{\pi}{N_x} + \cos \frac{\pi}{N_y}]^2}} - 1. \quad (7.64)$$

For $N_x = N_y = 9$, $\alpha \approx .49$.

EXERCISE 7.7

Solve the two dimensional heat equation for a 1 m^2 plate using a 9×9 grid, subject to the following boundary conditions:

- i) Along the left and bottom edges, the temperature is held fixed at 100° C .
- ii) Along the right and top edges, the temperature is held fixed at 0° C .

These boundary conditions admit a relatively simple solution, which can also be found analytically.

Isotherms

While the numerical output you've generated is correct and useful, it's rarely easy to recognize patterns in a collection of numerical results. Computational physics is moving away from purely numerical outputs toward presenting data, or results of calculations, in a graphical manner — scientific visualization. The state-of-the-art is far beyond the scope of this book, but surely we can do better than presenting a pile of numbers. For the heat problem, one possibility that comes to mind is to plot lines of constant temperature, or *isotherms*. Such contour plots are familiar to map readers everywhere. (*Isobars*, contours of equal barometric pressure, are often drawn on weather maps.) With a little practice, the interpretation of contour plots becomes fairly easy and a wealth of information becomes available.

The contouring algorithm we use is very simple — and always (?) works! (Some very sophisticated algorithms are known to produce erroneous contours in certain circumstances.) The subroutine is described in the Appendix; to use it, simply modify your code to include the following:

```

Subroutine HEAT2D
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This code fragment implements Successive Over-Relaxation
* to solve the 2D Heat equation on a 9 x 9 grid.
*
    ...
    character*1 response
    integer Nx,Ny,Ncl
    double precision contour(10), xx(10), yy(10)

    ...

    IF (.NOT.DONE)goto 100
*
* SOR is finished. Set up to draw contours...
*
    write(*,*)'SOR complete. Want to see the contours?'
    read(*,*)response
    IF( response .eq. 'y' .or. response .eq. 'Y')THEN
        call ginit
*
* Ncl = < number of contour lines to be drawn>
* contour(1) = < Temperature of contour 1 >

```

```

*   contour(2) = < Temperature of contour 2 >
*   ...
*   xx(1) = < x-coordinate of first grid line >
*   xx(2) = < x-coordinate of second grid line >
*   ...
*   yy(1) = < y-coordinate of first grid line >
*   yy(2) = < y-coordinate of second grid line >
*   ...
*   Nx = < number of grid points along x-axis >
*   Ny = < number of grid points along y-axis >
*
*   U( Nx, Ny ) = data to be plotted
*
      call CONTOURS ( Nx, Ny, hx, hy, u, Ncl, contour )

      call gend
ENDIF
...
end

```

EXERCISE 7.8

Modify your program so as to plot the isotherms after the calculation has been completed.

Irregular Physical Boundaries

It can happen, of course, that the physical plate doesn't match the grid very well. This can happen, for example, if we try to use a rectangular grid to investigate the temperature distribution in a circular plate. The best resolution of the difficulty is to use a grid more suitable to the problem. In the case of the circular plate, a cylindrical coordinate system is a much more natural choice than a Cartesian one. However, there are times when the problem is more than simply a choice of coordinate system. Consider, for example, a triangular plate — there's no coordinate system that makes that problem nice. We must face up to the possibility, nay, probability, that we cannot always choose our grid points to lie on the boundary of the object. For this case, we must come up with an acceptable procedure.

Actually, it's not all that difficult, once the need for special treatment along the boundary has been recognized. Consider the example shown in Figure 7.2. The function is known all along the boundary; the difficulty is just

that the boundary doesn't happen to lie on a grid line. The real problem is simply to express the second derivative in terms of the value at the boundary rather than at the node. As usual, we expand in a Taylor series about x_i :

$$f(x_i + \varepsilon) = f(x_i) + \varepsilon f'(x_i) + \frac{\varepsilon^2}{2} f''(x_i) + \frac{\varepsilon^3}{3!} f'''(x_i) + \cdots \quad (7.65)$$

and

$$f(x_i - h) = f(x_i) - hf'(x_i) + \frac{h^2}{2} f''(x_i) - \frac{h^3}{3!} f'''(x_i) + \cdots \quad (7.66)$$

Eliminating $f'(x_i)$ from these two equations and solving for $f''(x_i)$, we easily find the finite difference approximation

$$f''(x_i) = \frac{hf(x_i + \varepsilon) - (h + \varepsilon)f(x_i) + \varepsilon f(x_i - h)}{h\varepsilon(h + \varepsilon)/2} + \frac{h - \varepsilon}{3} f'''(x_i) + \cdots \quad (7.67)$$

This approximation to the second derivative can then be used in the expression for the Laplacian. We easily find that the analogue of Equation (7.63) is

$$u_{i,j} = \frac{\varepsilon}{2(h + \varepsilon)} \left[\frac{2h^2}{\varepsilon(h + \varepsilon)} u(x_i + \varepsilon, y_j) + \frac{2h}{h + \varepsilon} u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right]. \quad (7.68)$$

Depending upon how the physical boundary traverses the computational grid, appropriate finite difference approximations for various differential operators are easily obtained in a similar manner.

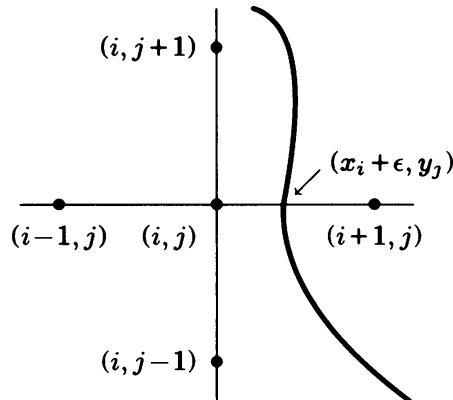


FIGURE 7.2 A boundary need not lie along a grid line.

EXERCISE 7.9

Using a Cartesian grid, solve for the temperature distribution on a circular plate. The temperature along one-quarter of the plate is held fixed at 100°C , while the remainder of the circumference is held at 0°C .

Neumann Boundary Conditions

So far, we've only discussed the case in which the *function* is known along the boundary, the *Dirichlet* boundary condition. There is an alternative boundary condition: if we know the *normal derivative* of the function at all points along the boundary, we have *Neumann* conditions. (There is yet another type of boundary condition, the *Cauchy* conditions, in which both the function and its normal derivative are known on the boundary. For Poisson's equation, Cauchy's condition is too restrictive — either Dirichlet or Neumann conditions are sufficient to generate a unique solution.) If the boundary lies along a grid line, the Neumann boundary condition is easily incorporated into our relaxation technique.

Let's reconsider our square metal plate. Only this time, instead of fixing the temperatures along the boundary, we'll fix the heat flow. That is, imagine that we are supplying heat to one edge, so there's a known flux of heat entering the plate along that edge. And let's further imagine that the heat flow out of the plate is different along the different edges, so that twice as much heat flows out the right edge as the top, and twice as much flows out the bottom as flows out the right. Since we are maintaining a steady-state temperature distribution, we must have that the total heat flowing into the square is equal to the amount flowing out — this is referred to as the compatibility condition. If this condition is not met, the plate cannot be in a steady state and its temperature will either rise or fall.

Let's impose an $N \times M$ grid on the metal plate, as in Figure 7.3, and consider the left side of the computational boundary for which $i = 0$. Laplace's equation for the points along this boundary is

$$\frac{u_{1,j} - 2u_{0,j} + u_{-1,j}}{h_x^2} + \frac{u_{0,j+1} - 2u_{0,j} + u_{0,j-1}}{h_y^2} = 0. \quad (7.69)$$

But with Neumann conditions, we also know the derivative along the bound-

ary,

$$\left. \frac{\partial u(x, y)}{\partial x} \right|_{x=0} = A_j \approx \frac{u_{1,j} - u_{-1,j}}{2h_x}, \quad (7.70)$$

where we've introduced A_j as the partial derivative at (x_0, y_j) . We've also introduced $u_{-1,j}$ into these expressions, which isn't even on the grid! This point is introduced to help us think about the problem, but is never actually used. Eliminating it yields an expression for the temperature on the boundary explicitly involving the boundary condition:

$$u_{0,j} = \frac{h_x^2 h_y^2}{2h_x^2 + 2h_y^2} \left[\frac{2u_{1,j} - 2h_x A_j}{h_x^2} + \frac{u_{0,j+1} + u_{0,j-1}}{h_y^2} \right], \quad 0 < j < M-1. \quad (7.71)$$

If $h_x = h_y = h$, this expression simplifies to

$$u_{0,j} = \frac{2u_{1,j} - 2hA_j + u_{0,j+1} + u_{0,j-1}}{4}, \quad 0 < j < M-1. \quad (7.72)$$

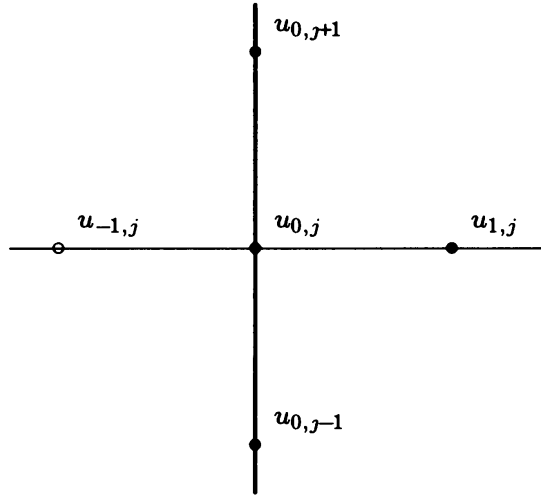


FIGURE 7.3 Applying Neumann conditions at a boundary. The open circle denotes a point off the computational grid.

In like fashion, we have along the right side

$$B_j = \left. \frac{\partial u(x, y)}{\partial x} \right|_{x=x_{N-1}} = \frac{u_{N,j} - u_{N-2,j}}{2h}, \quad 0 < j < M-1, \quad (7.73)$$

$$u_{N-1,j} = \frac{2u_{N-2,j} + 2hB_j + u_{N-1,j+1} + u_{N-1,j-1}}{4}; \quad (7.74)$$

along the bottom edge

$$C_i = \left. \frac{\partial u(x, y)}{\partial y} \right|_{y=y_0} = \frac{u_{i,1} - u_{i,-1}}{2h}, \quad 0 < i < N-1, \quad (7.75)$$

$$u_{i,0} = \frac{u_{i+1,0} + u_{i-1,0} + 2u_{i,1} - 2hC_i}{4}; \quad (7.76)$$

and along the top edge

$$D_i = \left. \frac{\partial u(x, y)}{\partial y} \right|_{y=y_{M-1}} = \frac{u_{i,M} - u_{i,M-2}}{2h}, \quad 0 < i < N-1, \quad (7.77)$$

$$u_{i,M-1} = \frac{u_{i+1,M-1} + u_{i-1,M-1} + 2u_{i,M-2} + 2hD_i}{4}. \quad (7.78)$$

The Laplacian at the corners of the computational grid will have two off-grid elements. Suitable approximations are found to be

$$u_{0,0} = (u_{0,1} - hC_0 + u_{1,0} - hA_0)/2, \quad (7.79)$$

$$u_{0,M-1} = (u_{0,M-2} + hD_0 + u_{1,M-1} - hA_{M-1})/2, \quad (7.80)$$

$$u_{N-1,0} = (u_{N-2,0} + hB_0 + u_{N-1,1} - hC_{N-1})/2, \quad (7.81)$$

$$u_{N-1,M-1} = (u_{N-2,M-1} + hB_{M-1} + u_{N-1,M-2} + hD_{N-1})/2. \quad (7.82)$$

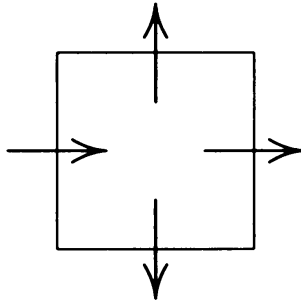


FIGURE 7.4 The direction of heat flow entering and leaving the square plate.

In order to have a flow of heat, there must be a temperature gradient. For our problem, let's take the gradients to be

$$A_j = -700^\circ\text{C/m},$$

$$B_j = -200^\circ\text{C/m},$$

$$C_i = +400^\circ\text{C/m},$$

$$D_i = -100^\circ\text{C/m}, \quad \text{for all } i \text{ and } j.$$

The heat will then flow as indicated in Figure 7.4. Note that if we evaluate the line integral of the *normal derivative* of the temperature around the perimeter of the square, we find that the net heat entering the plate is zero — the compatibility condition for a steady-state solution.

The astute reader will realize that we don't have enough information to determine the solution: we have only derivative information, which yields a family of solutions which satisfy both Laplace's equation and the Neumann boundary condition. That is, we know how the temperature *changes* from point to point, but we don't know what the actual temperature is. To establish a unique solution to our problem, we need to know the steady-state temperature at some point: let's imagine that a thermocouple on the lower left corner of the plate reads 750°C . This information is incorporated into the calculation by simply shifting all temperatures up (or down) so that the calculated temperature at the corner is correct. Mathematically, this shifting needs to be done only once, after the calculation has converged. However, you might want to shift with every iteration to help convey the progress the calculation is making toward convergence.

EXERCISE 7.10

Find the temperature everywhere on the plate. Plot isotherms at every 100°C to help visualize the temperature distribution.

A Magnetic Problem

Many, if not most, problems that you see as a student of physics are far removed from the problems seen by a physicist. This, of course, has a lot to do with the way we teach physics, the inherent difficulty of the subject, and the fact that *real* problems tax the abilities of Ph.D.'s and are simply not within the capabilities of an undergraduate student. However, the computer has a way of bridging the gulf between student and professor — an appropriate program can be used by the student to solve problems that can't otherwise be solved by anyone! The following is one such problem.

We want to learn about the magnetic field associated with a particular permanent magnet in the shape of a circular ring, as shown in Figure 7.5. Because of the magnet's shape, cylindrical coordinates are appropriate for the problem. However, since the magnet has finite extent along the z -axis, *the problem is not separable* and no analytic solution exists! The only possible solution is a numerical one, such as the one we now develop.

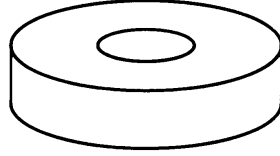


FIGURE 7.5 The professor's magnet, used to afix loose pieces of paper to his metal filing cabinet.

All electromagnetic phenomena, including the magnetic field produced by this magnet, are described by Maxwell's equations:

$$\nabla \cdot \mathbf{D} = \rho, \quad (7.83)$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \quad (7.84)$$

$$\nabla \cdot \mathbf{B} = 0, \quad (7.85)$$

and

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t} + \mathbf{J}. \quad (7.86)$$

Now, \mathbf{H} and \mathbf{B} are related by

$$\mathbf{B} = \mu_0(\mathbf{H} + \mathbf{M}), \quad (7.87)$$

where \mathbf{M} is the magnetization of the material. That is, the magnetic field \mathbf{B} is due to two factors: *macroscopic* currents and time-varying electric fields, which contribute to the magnetic intensity \mathbf{H} , and the inherent properties of the media, as manifest in \mathbf{M} . (We note that the permanent magnetism is due to the motion of subatomic particles on the *microscopic* scale within a magnetic domain.) For our purposes, we'll assume that the magnetization is homogeneous and is directed parallel to the axis of the magnet. For a steady-state problem, all the time derivatives vanish; there are no electric charges, so $\rho = 0$; and there are no currents, so that $\mathbf{J} = 0$. Thus

$$\nabla \times \mathbf{H} = 0. \quad (7.88)$$

Since the curl vanishes, we're naturally led to write

$$\mathbf{H} = -\nabla\Phi, \quad (7.89)$$

where Φ is a scalar magnetic potential. From Equation (7.85) we then have that the potential satisfies

$$\nabla^2\Phi = \nabla \cdot \mathbf{M}. \quad (7.90)$$

Since we've assumed that the magnet is homogeneous, the divergence $\nabla \cdot \mathbf{M}$ is zero within the magnet. Outside the magnet, the magnetization and its divergence are also zero. There is, of course, a discontinuity at the boundary that must be considered. So, except at the surface of the magnet, the potential is described by Laplace's equation.

Boundary Conditions

From our studies of first year physics, we know that at the interface between two media we must have

$$\hat{\mathbf{n}} \times (\mathbf{H}_1 - \mathbf{H}_2) = 0, \quad (7.91)$$

where $\hat{\mathbf{n}}$ is a unit normal vector at the interface. In terms of the scalar potential, we find that

$$\hat{\mathbf{n}} \times (\nabla \Phi_1 - \nabla \Phi_2) = 0. \quad (7.92)$$

Integrating *along* the interface yields the condition that

$$\Phi_1 = \Phi_2. \quad (7.93)$$

We also have a condition on the \mathbf{B} field:

$$\mathbf{B}_2 \cdot \hat{\mathbf{n}} - \mathbf{B}_1 \cdot \hat{\mathbf{n}} = 0, \quad (7.94)$$

which tells us that the *normal component* of \mathbf{B} is continuous. In terms of the potential, we then find

$$(-\nabla \Phi_1 + \mathbf{M}_1) \cdot \hat{\mathbf{n}} = (-\nabla \Phi_2 + \mathbf{M}_2) \cdot \hat{\mathbf{n}},$$

or

$$(\nabla \Phi_1 - \nabla \Phi_2) \cdot \hat{\mathbf{n}} = (\mathbf{M}_1 - \mathbf{M}_2) \cdot \hat{\mathbf{n}}. \quad (7.95)$$

This then is a condition on the gradient of the magnetic potential — as we cross the interface, the gradient changes so as to balance the change in the magnetization.

There is another boundary condition, that at infinity. Since the magnet is a dipole, the asymptotic form of the potential is

$$\Phi(\vec{r}) = -\frac{V}{4\pi} \mathbf{M} \cdot \nabla \left(\frac{1}{r} \right) = \frac{VMz}{4\pi r^3}. \quad (7.96)$$

This boundary condition can be treated in several ways, the simplest of which is to take a large grid and set the potential to zero on its boundaries. In electrostatic problems, where the potential often *is* zero on the boundary, this is certainly correct. In the present context, however, it's not such a wise choice. We'll use the asymptotic values determined by Equation (7.96) to set the values at the boundary of our computational grid, and solve the finite difference equations on this grid. Then we'll repeat the calculation with a larger grid to insure that the results are independent of the grid size.

The Finite Difference Equations

In circular cylindrical coordinates, the Laplacian is given as

$$\nabla^2\Phi = \frac{1}{\rho} \frac{\partial}{\partial\rho} \left(\rho \frac{\partial\Phi}{\partial\rho} \right) + \frac{1}{\rho^2} \frac{\partial^2\Phi}{\partial\phi^2} + \frac{\partial^2\Phi}{\partial z^2}. \quad (7.97)$$

Our problem possesses cylindrical symmetry and does not depend upon the angle ϕ , so that the potential is independent of ϕ . This is, of course, a great simplification. The magnet I have in mind is of the short and squat ring variety, the type used to hold notes against metal file cabinets in physics professors' offices, about 28 mm in diameter, 6 mm thick, and with a 10 mm hole in the middle. We will use a grid in ρ and z — for simplicity, let's choose the step size to be the same in both directions, and place the origin of the coordinate system at the geometrical center of the magnet. (See Figure 7.6.) We then have

$$\nabla^2\Phi \approx \frac{\Phi_{i+1,j} - \Phi_{i-1,j}}{2h\rho_i} + \frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{h^2} + \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{h^2}, \quad (7.98)$$

where we've introduced the notation

$$\Phi_{i,j} = \Phi(\rho_i, z_j), \quad 0 \leq i \leq N_\rho, \quad 0 \leq j \leq N_z. \quad (7.99)$$

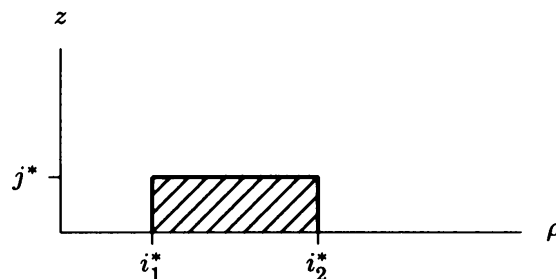


FIGURE 7.6 Cross section of the ring magnet in the first quadrant.

Because of the symmetry of the physical object, we shouldn't need to consider the potential other than in this quadrant. For a point not on the boundary of the magnet and with neither ρ nor z equal to zero, Equation (7.90) reduces to Laplace's equation and, with the above approximation for the Laplacian, we find that

$$\Phi_{i,j} = \frac{1}{4} \left[\left(1 + \frac{h}{2\rho_i}\right)\Phi_{i+1,j} + \left(1 - \frac{h}{2\rho_i}\right)\Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} \right], \quad i, j \neq 0. \quad (7.100)$$

Along the symmetry axis of the magnet, where $\rho = 0$, we must exercise some care; for our problem, the Laplacian is of the form

$$\nabla^2 = \frac{\partial^2 \Phi}{\partial \rho^2} + \frac{1}{\rho} \frac{\partial \Phi}{\partial \rho} + \frac{\partial^2 \Phi}{\partial z^2} \quad (7.101)$$

The second term looks like it could give some trouble as $\rho \rightarrow 0$. However, without a physical reason, you wouldn't expect a singularity in this problem. (Or any other problem, for that matter!) We thus suspect that there's an easy way out, and indeed there is, if we make the very plausible assumption that Φ is a continuous function of position. For constant z , the physical environment is the same in all directions as ρ approaches 0, so that Φ can't be changing; that is, the partial derivative with respect to ρ must go to zero as ρ goes to zero. Then we can use l'Hospital's rule to evaluate the term,

$$\left. \frac{1}{\rho} \frac{\partial \Phi}{\partial \rho} \right|_{\rho=0} = \left. \frac{\partial \Phi / \partial \rho}{\rho} \right|_{\rho=0} = \lim_{\rho \rightarrow 0} \frac{\frac{\partial}{\partial \rho}(\partial \Phi / \partial \rho)}{\frac{\partial}{\partial \rho}(\rho)} = \left. \frac{\partial^2 \Phi}{\partial \rho^2} \right|_{\rho=0}. \quad (7.102)$$

The correct expression for the Laplacian is thus

$$\nabla^2 \Phi = 2 \frac{\partial^2 \Phi}{\partial \rho^2} + \frac{\partial^2 \Phi}{\partial z^2}, \quad \rho = 0. \quad (7.103)$$

With this form for the Laplacian, the correct finite difference equation is found to be

$$\Phi_{0,j} = \frac{[4\Phi_{1,j} + \Phi_{0,j+1} + \Phi_{0,j-1}]}{6}. \quad (7.104)$$

The solution along $z = 0$ is even simpler. Our little magnet is a dipole, with a north end and a south end. Hence the solutions for $z < 0$ must be exactly opposite those for $z > 0$. That is, Φ must be an odd function of z , and so must be zero at $z = 0$. This is true at all points in the $z = 0$ plane,

$$\Phi_{i,0} = 0, \quad \text{for all } i. \quad (7.105)$$

We now move to the tougher boundary conditions, those at the physical surface of the magnet. The first thing to realize is that Laplace's equation is not valid here — the operative equation is Equation (7.90), which at the surface includes a consideration of the change of the magnetization. We first consider the cylindrical surfaces of the magnet at $\rho = 5$ and 14 mm. Since the magnetization has no component normal to the surface, Equation (7.95) tells us that the normal derivative is continuous. If the derivative just inside the surface is equal to the derivative just outside the surface, then the second derivative must be zero! The requirement is then

$$\left. \frac{\partial^2 \Phi}{\partial \rho^2} \right|_{i^*} = 0,$$

or

$$\Phi_{i^*,j} = \frac{\Phi_{i^*+1,j} + \Phi_{i^*-1,j}}{2}, \quad (7.106)$$

where i^* is a grid index for points lying on the cylindrical surface of the magnet.

Along the top edge, $z = 3$ mm, we must account for the sudden change in the magnetization. In particular, we have from Equation (7.95):

$$\left. \frac{\partial \Phi}{\partial z} \right|_{inside} - M = \left. \frac{\partial \Phi}{\partial z} \right|_{outside}. \quad (7.107)$$

Using forward and backward formulas to express the derivative inside and outside the magnet (at the surface), we find

$$\frac{\Phi_{i,j^*} - \Phi_{i,j^*-1}}{h} - M = \frac{\Phi_{i,j^*+1} - \Phi_{i,j^*}}{h}, \quad (7.108)$$

where j^* is the grid index for points lying on the physical surface. Solving for Φ_{i,j^*} , we find

$$\Phi_{i,j^*} = \frac{Mh + \Phi_{i,j^*+1} + \Phi_{i,j^*-1}}{2}. \quad (7.109)$$

There remains one problem, that of determining Φ_{i^*,j^*} . Thus far, we have two expressions for these points, Equations (7.106) and (7.109), and they don't agree. Forsaking more rigorous avenues of attack, we simply retreat and employ the average of the two values available:

$$\Phi_{i^*,j^*} = \frac{Mh + \Phi_{i^*,j^*+1} + \Phi_{i^*,j^*-1} + \Phi_{i^*+1,j^*} + \Phi_{i^*-1,j^*}}{4}. \quad (7.110)$$

Another Comment on Strategy

The general idea of what we need to do is pretty clear. However, the implementation of that idea can spell the difference between success and failure. Before you read any further, take a few minutes and think about how you would program this problem.

Have any ideas? It is tempting to just start coding, one special case after another, one DO LOOP at a time. But that's a dangerous approach, for many reasons: what happens when you change i^* ? Debugging becomes a nightmare. Besides, when all the cases are *special*, then none of them are unusual, are they? Can we instead develop a general approach that allows for differences from point to point while maintaining an evenhanded treatment of them? Such an approach will surely work as well as a collection of special cases, and will certainly be easier to write, debug, and modify.

We accomplish this by the introduction of an auxiliary array of flags. As part of the initialization, the “uniqueness” of each point is characterized by an index in the array FLAG. For the present problem, six indices will suffice. These correspond to points where:

- 1) Laplace's equation holds, and the solution is given by Equation (7.100);
- 2) the point lies on the boundary of the computational grid, and the solution is fixed by boundary conditions;
- 3) $\rho = 0$, and the solution is given by Equation (7.104);
- 4) the point is on the cylindrical wall, the solution is given by Equation (7.106);
- 5) the point is on the top surface, with the solution given by Equation (7.109); and
- 6) the point is (i^*, j^*) , the solution is given by Equation (7.110).

At initialization, these indices are stored in the flag array and the solution array is initialized, including boundary conditions.

You're almost ready to begin developing code to solve this problem, except for one obstacle that we've encountered before — units. Since the computer stores only numbers, and not units, it is imperative that you understand the units and dimensions of the problem, and that all variables and parameters are expressed correctly. These difficulties are particularly acute

in magnetic phenomena, since many of us do not have an intuitive feel for the units that are commonly employed.

In SI units, B is given in Tesla, H and M in Ampere meter⁻¹, and μ_0 has the value $4\pi \times 10^{-7}$ Tesla Ampere⁻¹ meter. (For reference, the Earth's magnetic field is on the order of 6×10^{-5} Tesla.) Typical values of the magnetization for permanent magnets are in the $10^5 - 10^6$ range, so we'll take the magnetization of our magnet to be 10^5 Ampere meter⁻¹. Now, as long as we remember to express all distances in meters, we should be ready to continue.

You're now ready to start developing a computer code to solve this problem. Begin with the initialization; where better to start than the beginning? Assume that the solution is sought on a 25×25 grid at 1 mm spacing, so that $i_1^* = 6$, $i_2^* = 15$, and $j^* = 4$. Not having any particular insight at this point, initialize the grid to unity. Then initialize the solution array for distances far from the magnet, using Equation (7.96), and for all other "special" places. As you sweep through the array, initialize both the flag and the solution arrays so that the connection between the two is clear. For example, as you set the solution matrix to its asymptotic value along the far right boundary, set the corresponding element in the flag array to "2," indicating that it shouldn't be allowed to relax, e.g., it's a fixed value.

After having put this effort into the initialization, the main relaxation code becomes relatively simple. We note the effort spent in organizing our approach and initializing the arrays is effort well spent — it will make the subsequent program much easier to write and to understand. The actual relaxation process is then clear-cut: as we move through the solution array, we query the flag array to determine how that particular point is to be treated. The computer code for the problem might look something like the following:

```

Subroutine Ring_Magnet
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine solves for the scalar potential in the
* cylindrical magnet problem, by the method of SOR.
*
*
*                               May 1, 1966
*
      double precision pi, m, PHI(25,25), current, ...
      integer flag(25,25)
      logical done
      parameter (m = 1.0D5, pi=3.14159265358979D0)
      ...

```

```

*
* FLAG is an integer array that characterizes a point
* in the grid. In particular, it is used to determine
* which of various equations to use in determining the
* solution, according to the following "code":
*
* 1. typical point, use Equation (7.100).
* 2. fixed value point, either on the boundary of the
*    computational grid or determined by symmetry.
*    NOT SUBJECT TO CHANGE!
* 3. along the axis of the magnet, use Equation (7.104).
* 4. on the cylindrical walls, use Equation (7.106).
* 5. on the top surface, use Equation (7.109).
* 6. at a corner, use Equation (7.110).
*
* Initialize PHI and FLAG:
*
*      DO i = 1, imax
*        DO j = 1, imax
*          flag(i,j)=1
*          phi(i,j) = 1.d0
*        END DO
*      END DO
*      ...
* Initialize all boundaries, points, etc.
*
*      DO j = 1, jstar-1
*        flag(istar,j) = 4
*      END DO
*      ...
* Initialize other variables:
*      h = ...
*      Nx = ...
*      Ny = ...
*
* Use optimum alpha for 2D Cartesian coordinates.
* Although not optimum for this problem, it should be
* a reasonable guess.
*
*      alpha = ...
*      CALL CLEAR
*      ...
*      count = 0
*

```

```

*   SOR iteration starts here...
*
100   DONE = .TRUE.
      count = count + 1
      IF (count .GE. 200) stop 'Too many iterations!'
*
*   The following DO LOOPS do the actual SOR, using the
*   relaxation parameter alpha:
*           "phi(i,j)" is the old value,
*           "current" is the latest iterate.
*
      DO i = 1, Nx
        DO j = 1, Ny
          ...
          IF ( flag(i,j) .eq. 1 )THEN
            current = ...
          ELSE IF(flag(i,j) .eq. 2)THEN
            current = ...
          ...
          ELSE
            stop ' Illegal value for flag!'
          ENDIF

          IF (abs((current-phi(i,j))/current).gt.5.d-4)
+         DONE = .FALSE.
          phi(i,j) = current + alpha * (current-phi(i,j))
        END DO
      END DO

      call cursor(1,1)
      ...
1000  format(20f4.1)
      IF ( .NOT. DONE ) goto 100
      ...
end

```

Note that the computer code contains all the documentation concerning the values of FLAG, so you understand how (and why) different points of the grid are to be treated. This variable treatment is handled in a very straightforward manner in the main SOR loop, which examines the value of FLAG(I, J) and acts upon PHI(I, J) accordingly. If we should later need to change the geometry, include a larger grid, or make other changes, the modifications can be made in a relatively easy manner. In order to continue to write updates to the screen as the calculation progresses, the format has been changed sub-

stantially — of course, *final results* can then be written in whatever format deemed appropriate. You might also want to display lines of constant potential, or even $|B|$.

■ EXERCISE 7.11

Using a magnetization of 10^5 Ampere meter $^{-1}$, calculate the magnetic potential on the 25×25 grid we've been discussing. Final results should be displayed to 4 places.

Are We There Yet?

Having the computer supply us with numbers is not the same thing as having solved the problem, of course. The results of this calculation depend upon the size of the grid — after all, 2.5 cm is a long way from infinity. While the iterations continued until $\Phi_{i,j}$ was converged to 4 significant digits, this *does not* mean that the results are accurate to 4 places. We must still vary the size of the grid, and see how the results might change. We might also want to change the grid spacing. In general, all the parameters of the calculation need to be tested to insure the convergence of the results.

EXERCISE 7.12

Vary the parameters of the calculation so as to achieve 5% accuracy in the calculation of the magnetic potential.

Spectral Methods

With finite difference methods, progress toward an approximate solution proceeds in a rather direct fashion. But sometimes there are advantages to being indirect. Previously, we've seen that the Fourier transform, particularly as implemented in the FFT algorithm, can be the basis of extremely accurate and efficient methods of solution. Let's see how it might be used here.

The basic idea of spectral methods is to *transform* the problem from its original description into one involving the Fourier transform of the solution. The motivation is that in many situations the solution in the transform space is much easier to obtain than is the solution in the original space. (Of course, if the solution isn't easier, we wouldn't be using this method!) The solution

to the original problem is then obtained by evaluating the inverse Fourier transform of the solution found in transform space.

A simple example might be worthwhile. Consider the diffusion equation

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}, \quad (7.111)$$

which describes the temperature $T(x, t)$ in a long metal rod. We begin at time $t = 0$ knowing the initial distribution of temperature, $T(x, 0)$. The problem is to find the temperature along the rod at later times. If $\tau(k, t)$ is the Fourier transform of $T(x, t)$, then

$$\tau(k, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} T(x, t) e^{-ikx} dx \quad (7.112)$$

and we can write the temperature as

$$T(x, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tau(k, t) e^{+ikx} dk. \quad (7.113)$$

Note that we are transforming between x and k at the same instant in time. Substituting this expression for the temperature into our diffusion equation, we find

$$\frac{\partial}{\partial t} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tau(k, t) e^{+ikx} dk = \kappa \frac{\partial^2}{\partial x^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tau(k, t) e^{+ikx} dk. \quad (7.114)$$

On the left side, the partial derivative operates on $\tau(k, t)$. But $\tau(k, t)$ is not a function of x , so that on the right side the derivative operates only on the exponential factor, to yield

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{\partial \tau(k, t)}{\partial t} e^{+ikx} dk = \kappa \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tau(k, t) [-k^2] e^{+ikx} dk. \quad (7.115)$$

We now multiply both sides by $e^{-ik'x}$, and integrate over x to find

$$\begin{aligned} & \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ik'x} \int_{-\infty}^{\infty} \frac{\partial \tau(k, t)}{\partial t} e^{+ikx} dk dx \\ &= \kappa \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ik'x} \int_{-\infty}^{\infty} \tau(k, t) [-k^2] e^{+ikx} dk dx. \end{aligned} \quad (7.116)$$

Exchanging the order of the integration and using the expression

$$\int_{-\infty}^{\infty} e^{i(k-k')x} dx = 2\pi \delta(k - k') \quad (7.117)$$

for the Dirac delta $\delta(k - k')$, we find

$$\frac{\partial \tau(k, t)}{\partial t} = -k^2 \kappa \tau(k, t). \quad (7.118)$$

In the original statement of the problem, Equation (7.111), we had two partial derivatives — we now find that in our “transform space,” we have only one. Its solution is easily found to be

$$\tau(k, t) = e^{-k^2 \kappa t} \tau(k, 0), \quad (7.119)$$

where $\tau(k, 0)$ is obtained by transforming the initial temperature distribution,

$$\tau(k, 0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} T(x, 0) e^{-ikx} dx. \quad (7.120)$$

Having determined $\tau(k, t)$, we can use Equation (7.113) to perform the inverse transform to find the temperature,

$$\begin{aligned} T(x, t) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tau(k, t) e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-k^2 \kappa t} \tau(k, 0) e^{+ikx} dk. \end{aligned} \quad (7.121)$$

We see, then, that the general approach consists of three steps: 1) transform the original partial differential equation, and the initial conditions, into transform space; 2) solve the (presumably simpler) equation in transform space; and 3) perform the inverse transform, so as to obtain the solution in terms of the original variables.

EXERCISE 7.13

Find the temperature in the long metal rod, and plot it, as a function of time. Take the initial temperature to be

$$T(x, 0) = \begin{cases} 0, & |x| > 1 \text{ m}, \\ 100^\circ\text{C}, & |x| \leq 1 \text{ m}, \end{cases}$$

and $\kappa = 10^3 \text{ m}^2/\text{sec}$.

Clearly, the advantage of the spectral approach is the ease with which the derivative can be treated in the transformed set of coordinates. For a number of problems, such as our diffusion example, this approach works very well. Of course, part of the reason it worked so well was that the equation was relatively simple — in transform space, our partial differential equation in two variables became a differential equation in one variable. In other cases, such a great simplification might not occur. In those instances, it might be advantageous to have the flexibility to treat part of the problem one way, and another part a different way.

The Pseudo-Spectral Method

Let's consider a different problem, the quantum mechanical scattering of a wavepacket from a step barrier. In many standard textbooks, this problem is discussed and figures are displayed which depict the time evolution of the wavepacket as it moves toward the barrier and interacts with it. But with the computer, we can calculate the relevant wavefunctions and watch the interaction. In one dimension, the Schrödinger equation is

$$\begin{aligned} i\hbar \frac{\partial \psi(x, t)}{\partial t} &= -\frac{\hbar^2}{2m} \frac{\partial^2 \psi(x, t)}{\partial x^2} + V(x)\psi(x, t) \\ &= (\mathbf{T} + \mathbf{V})\psi(x, t), \end{aligned} \tag{7.122}$$

where m is the mass of the particle, \mathbf{T} is the kinetic energy term, and \mathbf{V} is the potential. In quantum mechanics, both \mathbf{T} and \mathbf{V} are considered operators, although \mathbf{T} is clearly a derivative operator — $\mathbf{T}\psi$ is obtained by taking the derivative of ψ — while \mathbf{V} is simply a function that multiplies ψ .

We could, of course, solve this problem by finite differences. However, the discussion of the spectral method demonstrated that there is another, sometimes better, way to treat the derivative terms. In the present problem, the transform method would clearly enable us to treat the kinetic energy operator, but it would not help with the potential term. What we would really like is to treat the derivative term one way and the potential term another. In particular, we would like to treat the potential term in coordinate space, where it's simply a function, and treat the derivative term in transform space where it's easily (and accurately) evaluated.

Let's return to the Schrödinger equation. With regard to its time dependence, we can immediately write a *formal solution* to the equation. (On a philosophical note, solutions that are easily written are almost never very

useful. In this case, however, the formal solution will guide us to an excellent practical method for computationally solving the Schrödinger equation.) First we define the exponential of an operator as

$$e^{\mathbf{A}} = 1 + \mathbf{A} + \frac{1}{2!}\mathbf{A}\mathbf{A} + \frac{1}{3!}\mathbf{A}\mathbf{A}\mathbf{A} + \cdots \quad (7.123)$$

This mathematical definition is valid for operators in general, including the kinetic and potential energy operators which we have. With this definition, a formal solution to Equation (7.122) is

$$\psi(x, t) = e^{-i(\mathbf{T}+\mathbf{V})(t-t_0)/\hbar} \psi(x, t_0). \quad (7.124)$$

EXERCISE 7.14

Using the definition of the exponentiation of an operator, verify that Equation (7.124) is a solution to the Schrödinger equation, Equation (7.122).

For convenience, we define $\delta_t = t - t_0$, so that the *time evolution* of the wavefunction is given by

$$\psi(x, t) = e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} \psi(x, t_0). \quad (7.125)$$

This equation gives us a prescription for calculating $\psi(x, t)$ at any time t from knowledge of ψ at time t_0 . Furthermore, the prescription involves the exponential of the sum of our two operators. It's *extremely tempting* to write

$$e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} = e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/\hbar}. \quad (7.126)$$

Unfortunately, this expression is untrue! But if it were true, then we would have succeeded in splitting the time evolution operator into two terms, and could have treated each factor separately. As we will shortly see, while this equality is not valid, the decomposition can serve as a useful *approximation*. This technique of separating the evolution operator into two factors is generally termed the *split-operator approach*, and is more widely applicable than we have space to demonstrate.

Why is Equation (7.126) invalid? In general, derivatives and other operators yield different results when applied in a different order. For example, consider the operators $\mathbf{A} = x$ and $\mathbf{B} = d/dx$. Then

$$\mathbf{A}\mathbf{B}\psi = x \frac{d\psi}{dx}, \quad (7.127)$$

but

$$\mathbf{BA}\psi = \frac{d}{dx}(x\psi) = \psi + x\frac{d\psi}{dx} = (\mathbf{AB} + 1)\psi. \quad (7.128)$$

Since the order of the operators *does* matter, the operators *do not commute*. The commutability of operators — or, as the case may be, lack thereof — is expressed by the *commutator* of \mathbf{A} and \mathbf{B} . This is defined as

$$[\mathbf{A}, \mathbf{B}] = \mathbf{AB} - \mathbf{BA}, \quad (7.129)$$

and is evaluated by allowing it to act on an arbitrary function. (If the order of the operators is irrelevant, then the operators commute and the commutator is zero.) For our operators, we have

$$\left[x, \frac{d}{dx}\right]\psi = x\frac{d}{dx}\psi - \frac{d}{dx}x\psi = -\psi. \quad (7.130)$$

Since ψ is arbitrary, the commutator expresses a relationship between the operators x and d/dx independent of ψ ; hence we write

$$\left[x, \frac{d}{dx}\right] = -1. \quad (7.131)$$

We are now prepared to consider the product of two exponentials, such as $e^{\mathbf{A}}e^{\mathbf{B}}$. Expanding each of the factors according to the definition of the exponential of an operator, we can verify that

$$e^{\mathbf{A}}e^{\mathbf{B}} = e^{\mathbf{C}} \quad (7.132)$$

if and only if

$$\mathbf{C} = \mathbf{A} + \mathbf{B} + \frac{1}{2}[\mathbf{A}, \mathbf{B}] + \cdots. \quad (7.133)$$

This is a statement of the famous Baker–Campbell–Hausdorff theorem.

EXERCISE 7.15

Find the next two terms of \mathbf{C} .

As an exact expression, the Baker–Campbell–Hausdorff theorem tells us that Equation (7.126) is invalid unless \mathbf{T} and \mathbf{V} commute — which they don't (usually)! But more than that, the theorem is extremely useful in developing approximations of known error. For example, we now know that Equation (7.126), which simply ignores the commutator, is accurate through $\mathcal{O}(\delta_t)$. A

more accurate approximation is obtained by using a symmetric decomposition of the product, e.g., by using the expression

$$e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} \approx e^{-i\mathbf{V}\delta_t/2\hbar} e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/2\hbar}, \quad (7.134)$$

which is accurate through $\mathcal{O}(\delta_t^2)$.

We are now ready to describe an efficient method of solving the quantum mechanical scattering problem, using a mixture of coordinate space and transform space techniques called the *pseudo-spectral method*. Let's imagine that we know $\psi(x, t_0)$ — that is, we have an array containing $\psi_j = \psi(x_j, t_0)$ at the grid points x_j at the time t_0 . Ultimately, we will use the FFT to evaluate the necessary transforms so that the number of points on the grid will be a power of two. Recall that the total range in x determines the grid spacing in k , and *vice versa*. ψ at time t is approximated by

$$\psi(x, t) \approx e^{-i\mathbf{V}\delta_t/2\hbar} e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/2\hbar} \psi(x, t_0). \quad (7.135)$$

Since V is a function it is easily evaluated at the x_j . We can define an intermediate quantity $\phi(x)$ as

$$\phi(x) = e^{-i\mathbf{V}\delta_t/2\hbar} \psi(x, t), \quad (7.136)$$

and evaluate it on the grid as

$$\phi(x_j) = e^{-iV(x_j)\delta_t/2\hbar} \psi(x_j, t_0). \quad (7.137)$$

This evaluation is perfectly straightforward: in coordinate space, it's a simple sequence of arithmetic operations. In fact, if δ_t is constant throughout the calculation, an array of these exponential factors can be evaluated once, stored, and used whenever needed without reevaluation.

The next step is to determine the result of the exponential of the kinetic energy operating on ϕ ,

$$e^{-i\mathbf{T}\delta_t/\hbar} \phi(x_i).$$

In coordinate space, this is a troublesome term. It contains the second derivative operator, exponentiated! But in transform space, the derivative is easy to evaluate, as we've seen. To evaluate the term, we'll need $\Phi(k)$, the Fourier transform of $\phi(x)$,

$$\Phi(k) = \mathcal{F}[\phi(x)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(x) e^{-ikx} dx, \quad (7.138)$$

and the inverse relation

$$\phi(x) = \mathcal{F}^{-1}[\Phi(k)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) e^{+ikx} dk. \quad (7.139)$$

Then

$$\begin{aligned} e^{-i\mathbf{T}\delta_t/\hbar} \phi(x) &= e^{-i\mathbf{T}\delta_t/\hbar} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) e^{+ikx} dk \\ &= \left[1 + \frac{-i\mathbf{T}\delta_t}{\hbar} + \frac{(-i)^2 \mathbf{T}\mathbf{T}\delta_t^2}{2!\hbar^2} + \dots \right] \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) \left[1 + \frac{-i\mathbf{T}\delta_t}{\hbar} + \frac{(-i)^2 \mathbf{T}\mathbf{T}\delta_t^2}{2!\hbar^2} + \dots \right] e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) \left[1 + \frac{-i - \hbar^2 \delta_t}{\hbar} \frac{d^2}{2m dx^2} + \frac{1}{2!} \left(\frac{-i - \hbar^2 \delta_t}{\hbar} \right)^2 \frac{d^4}{dx^4} + \dots \right] e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) \left[1 + \frac{-iT(k)\delta_t}{\hbar} + \frac{1}{2!} \left(\frac{-iT(k)\delta_t}{\hbar} \right)^2 + \dots \right] e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Phi(k) \left[e^{-iT(k)\delta_t/\hbar} \right] e^{+ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-iT(k)\delta_t/\hbar} \Phi(k) e^{+ikx} dk \\ &= \mathcal{F} \left[e^{-iT(k)\delta_t/\hbar} \Phi \right], \end{aligned} \quad (7.140)$$

where we've introduced the kinetic energy term

$$T(k) = \frac{\hbar^2 k^2}{2m}. \quad (7.141)$$

The wavefunction at time t is then obtained by a final multiplication by the potential factor. The full propagation is thus given as

$$\psi(x, t) \approx e^{-iV(x)\delta_t/2\hbar} \mathcal{F} \left[e^{-iT(k)\delta_t/\hbar} \mathcal{F}^{-1} \left[e^{-iV(x)\delta_t/2\hbar} \psi(x, t_0) \right] \right]. \quad (7.142)$$

Although this seems a bit complicated, it has a certain elegance to it that is very attractive. The algorithm skips to and fro, to transform space and back again, weaving a solution from coordinate and transform space contributions. Yet the coding is very straightforward and easy to accomplish. And if the FFT is used to perform the Fourier transforms, it's extremely efficient and accurate as well. A general outline of the program will look like the following:

```

Subroutine WavePacket
*-----
* Paul L. DeVries, Department of Physics, Miami University
*
* This subroutine calculates the time evolution of a quantum
* mechanical wavepacket by the pseudo-spectral method.
*
*
* January 1, 1993
*
*      Complex*16 psi(256), phi(256)
*      double precision ...
*      integer ...
*
* Initialize the wavefunction:
*
*      DO i = 1, n
*          psi(i) = ...
*      END DO
*
* Initialize the arrays ExpT and ExpV,
*
* ExpT is an array of values  $\exp(-i T \text{ delta\_t} / \hbar)$ 
* ExpV is an array of values  $\exp(-i V \text{ delta\_t} / 2\hbar)$ 
*
* Initialize some other stuff:
*
*      dt = 5.0d-18                ! time step in seconds
*
* Loop over propagation steps until time > MaxTime:
*
*      Time = 0.d0
1000  time = time + dt
*
* First Step: multiply by exponent of V:
*
*      DO i = 1, N
*          phi(i) = ExpV(i) * psi(i)
*      END DO
*
* Second Step: multiply by exponent of T, by
* a) transform the wavefunction,
*
*      call fft( phi, m, 0 )
*

```

```

* b) multiply by ExpT,
*
      DO i = 1, n
        phi(i) = ExpT(i) * phi(i)
      END DO
*
* c) inverse transform,
*
      call fft (phi, m, 1 )
*
* Third and final step, multiply by exponent of V, again:
*
      DO i = 1, n
        psi(i) = ExpV(i) * phi(i)
      END DO
*
* Plot  psi**2
*
      ...
*
* Check if done:
*
      IF ( time .lt. MaxTime )goto 1000
*
      end

```

That's all there is to it! Given that we're solving a rather difficult problem, the simplicity of the basic algorithm is rather amazing.

What should we use as an initial wavefunction? One particularly interesting example is the Gaussian function

$$\psi(x, 0) = \frac{1}{\sqrt[4]{2\pi(\Delta x)^2}} e^{ik_0 x - \frac{(x-x_0)^2}{4(\Delta x)^2}}. \quad (7.143)$$

This is a plane wave, modulated by a Gaussian weighting function centered about $x = x_0$. The quantity Δx is related to the width of the function — if the width is small, the magnitude of the function quickly falls off away from x_0 . (Remember that in quantum mechanics it's the square of the wavefunction that is physically meaningful. Δx is the half-width of the square of the wavefunction, as measured between the points at which it has fallen to $1/e$ of its maximum value.)

In general, Heisenberg's uncertainty principle states that

$$\Delta x \Delta p \geq \frac{\hbar}{2}, \quad (7.144)$$

or, since $p = \hbar k$, that

$$\Delta x \Delta k \geq \frac{1}{2}. \quad (7.145)$$

The Fourier transform of the Gaussian function of Equation (7.143) is easily found to be

$$\phi(k, 0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \psi(x, 0) e^{-ikx} dx = \sqrt{\frac{4(\Delta x)^2}{2\pi}} e^{-(\Delta x)^2(k-k_0)^2}, \quad (7.146)$$

another Gaussian function. By comparing with Equation (7.143), we find that this function has a width $\Delta k = 1/(2\Delta x)$, or that

$$\Delta x \Delta k = \frac{1}{2}. \quad (7.147)$$

That is, while the product is always greater than or equal to $1/2$, we find that the equality holds for the Gaussian wavefunction — this is the reason it's also known as the *minimum uncertainty wavepacket*.

Next, we need to initialize the arrays ExpV and ExpT. The first is not difficult; merely evaluate the potential at the specified grid points and fill the array appropriately. The second is more of a problem, although it's not all that difficult. δ_k and δ_x are related by

$$\delta_k = \frac{2\pi}{N\delta_x}. \quad (7.148)$$

With N points, k appears to range from 0 to $(N-1)\delta_k$. But recall from Chapter 6 that the fast Fourier transform algorithm imposes a periodicity to the functions, so that k 's above the Nyquist frequency are actually negative. Since we need these k 's in order to evaluate the kinetic energy, we must be a little careful. The array ExpT can be properly filled with the following code:

```

Parameter (hbar2 = 7.6199682d0) !      hbar**2
...
DO i = 1, N/2
*
* For k's between 0 and the Nyquist cutoff:
*
      k      = dble(i-1) * delta_k
```

```

*
*   KE is the Kinetic Energy in eV
*
      KE      = k * k * hbar2 / ( 2.d0 * mass )
      ExpT(i) = cdexp( -im * KE * delta_t / hbar )
*
* K's above the Nyquist cutoff are actually negative:
*
      k        = -dble(i) * delta_k
      KE        = k * k * hbar2 / ( 2.d0 * mass )
      ExpT(N+1-i) = cdexp( -im * KE * delta_t / hbar )
END DO

```

A Sample Problem

Let's consider a specific example, a stationary wavepacket evolving in free space. That is, we'll set $k_0 = 0$ and $V(x) = 0$. As in Chapter 2, we'll find it convenient to use the parameter

$$\hbar^2 = 7.6199682 \, m_e \, \text{eV} \, \text{\AA}^2, \quad (7.149)$$

although we also know that $\hbar = 6.5821220 \times 10^{-16} \, \text{eV} \cdot \text{sec}$ and may find this expression useful as well. We'll take the initial wavepacket to be a Gaussian function located at $x_0 = 0$ and having a width of $\Delta x = 1 \, \text{\AA}$. As discussed in Chapter 6, the choices of grid spacing in the x and k coordinates and the number of points to be used in the calculation are not independent of one another. Let's tentatively choose the total range of the x -coordinate to be $20 \, \text{\AA}$, centered about $x = 0$, and use $N = 64$. This gives a grid spacing $\approx 0.31 \, \text{\AA}$, which should be adequate for a wavepacket this broad. With these parameters, we can propagate the wavefunction and find results such as presented in Figure 7.7.

Hint # 1: Choose δ_x to be sufficiently small so that all the interesting detail of the wavefunction is adequately sampled.

For times less than about 6×10^{-16} seconds, we observe a standard characteristic of quantum mechanical wavepackets — as time evolves, they spread out in space. For simple cases such as this one, an analytic result can be obtained for Δx as a function of time, which we could verify if we choose. Of more immediate interest to us, however, is the behavior of the wavepacket at even longer times. Physically, we would expect the wavepacket to continue to spread. Computationally, however, the wavepacket already nearly fills the

computational grid. Remember, the FFT imposes periodicity on the wavefunction — although we tend to view the solution as spreading toward $\pm\infty$, the computational reality is that the left and right extremes of the grid are the same point! Thus, as some portion of the wavepacket moves off the right edge of the grid, it *wraps around* and reappears on the left edge of the grid! But the wavefunction already exists at the left edge — the two contributions interact with one another and create an interference pattern. We should stress that this is NOT a physically meaningful result — our computational grid is *too small* to describe the physical situation adequately at these long times.

Hint # 2: Always use a grid large enough so that the physics occurs far from any edge.

The “cure” for this *wraparound problem* is to increase the size of the computational grid. However, δ_x shouldn’t be increased, since this was chosen so as to sample the wavefunction adequately. The only parameter remaining for us to change is N .

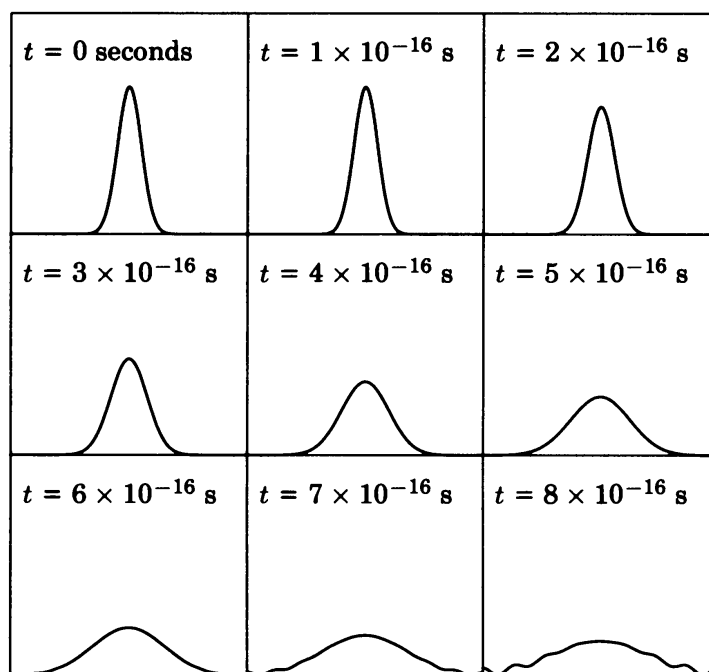


FIGURE 7.7 Time evolution of the Gaussian wavepacket discussed in the text.

EXERCISE 7.16

Use a time step of 5×10^{-18} seconds and replicate the results of Figure 7.7. Then try $N = 128$, to eliminate the wraparound problem.

Now let's see if the wavefunction can be made to move. All the dynamics that actually cause the wavepacket to move are already present in the program, so that all that needs to be changed is the initial value of k_0 . It's often convenient to use energy rather than wavenumber in specifying the initial conditions. For free particle motion, they're related through the relation

$$k = \sqrt{\frac{2mE}{\hbar^2}}. \quad (7.150)$$

EXERCISE 7.17

Let the initial kinetic energy be $E_0 = 150$ eV. Follow the evolution of the wavepacket for 100 time steps, with $\delta_t = 5 \times 10^{-18}$ seconds.

In addition to spreading, the wavefunction moves to the right. If not halted, it also wraps around, appearing at the left of the computational grid after moving off the right edge. Since the entire wavepacket is in motion, there's nothing for it to interfere with and so its shape is not distorted (beyond its natural spreading). Still, a cautious program would halt and inform the user that the wavepacket is nearing an edge. Let's try this again, with a larger kinetic energy.

EXERCISE 7.18

Repeat the exercise, with $E_0 = 300$ eV. What happened?

So far, we've concentrated on the wavefunction in coordinate space. But it's just as valid to ask what the wavefunction is doing in k -space. And in this case, particularly illuminating. Modify the code so that the probability distributions in coordinate space and in k -space are displayed simultaneously, and repeat the previous two exercises. You should see that in Exercise 7.16, the k -space wavefunction is entirely contained within the range from 0 to the Nyquist frequency. But in Exercise 7.18, the wavefunction extends beyond the Nyquist limit. Those components of the wavefunction at large positive frequencies must be included in order to describe this specific wavefunction accurately, but as far as the FFT is concerned, frequencies above the Nyquist limit are *negative*. This is essentially another *wraparound problem*, this time in k -space.

Hint # 3: Choose the grid in k -space sufficiently large so that all the interesting physics is contained well within the Nyquist limit. (That is, choose δ_x to be sufficiently small. This is really just Hint # 1, expressed in different language.)

EXERCISE 7.19

Modify the code as indicated and repeat the previous two exercises.

We see that the use of the spectral method is not without its difficulties. In particular, we must be very aware of the wraparound problem, in both its disguises. On the positive side, however, it's a problem that is very easy to diagnose — when it's not working properly, the method yields results that are obviously incorrect when plotted. Wraparound is also easy to monitor: simply determine how much of the wavefunction is in the last, say, 5% of the computational grid in coordinate space, and within 5% of the Nyquist limit in the reciprocal space. If the solution ever becomes appreciable in these regions, say, more than 0.5%, a warning message can be printed and the execution of the program terminated. We emphasize that when the method is used properly and within its domain of validity, it works extremely well, yielding accurate results in a very timely manner. In these situations, virtually no other method is capable of producing results of such accuracy in a comparable amount of time.

The Potential Step

Let's consider the behavior of a wavepacket at a step. Instead of being zero everywhere, the potential $V(x)$ is chosen to be

$$V(x) = \begin{cases} 0, & x \leq 0, \\ V_0, & 0 \leq x. \end{cases} \quad (7.151)$$

As before, we'll consider x on the range $-20 \text{ \AA} \leq x \leq 20 \text{ \AA}$, with $N = 128$. Let's take $E_0 = 100 \text{ eV}$ and $V_0 = 200 \text{ eV}$.

EXERCISE 7.20

Evolve the wavefunction for 50 time steps, $\delta_t = 5 \times 10^{-18} \text{ sec}$, using these parameters.

As the wavepacket collides with the step, the reflected components interfere with the unreflected ones, giving rise to a probability distribution that resembles an interference pattern, as indicated in Figure 7.8. It is at times such as this that the grid size becomes particularly important — δ_x must be small enough to sample all the detail of the wavefunction, *particularly* when the wavefunction is rapidly changing. After sufficient time, all the

components are reflected by the barrier and the wavepacket regains its initial shape.

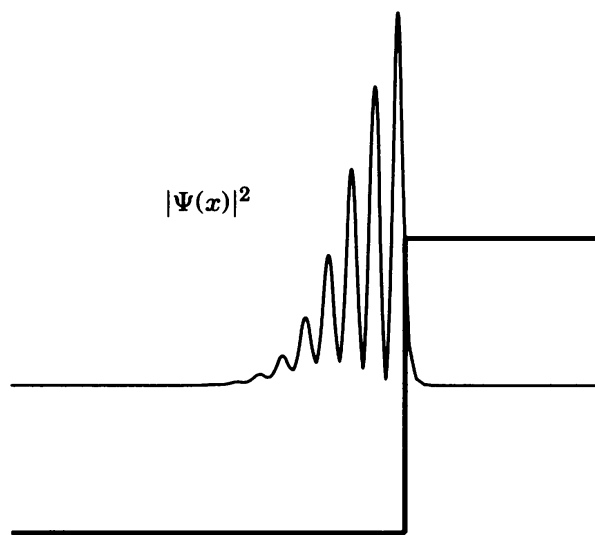


FIGURE 7.8 The probability distribution at some instant during the collision of the 100 eV wavepacket with the 200 eV step, on the range $-10 \text{ \AA} \leq x \leq 5 \text{ \AA}$. Note that the probability distribution extends *into* the step, a classically forbidden region.

We might want to increase the range, so that the evolution can be followed a bit longer in time. But if we increase the range and leave N the same, we are also increasing δ_x , which in turn decreases the range in k -space. Thus we need to increase N as well. A nice balance can be achieved by doubling N and increasing the range in x by $\sqrt{2}$. This has the effect of increasing the k range by a factor of $\sqrt{2}$ as well, while decreasing the grid spacing in both x - and k -coordinates.

EXERCISE 7.21

Evolve the wavefunction again, with $-30 \text{ \AA} \leq x \leq 30 \text{ \AA}$ and $N = 256$.

According to classical mechanics, the particle can never be found in a region in which the potential energy is greater than the total energy. As seen in Figure 7.8, in quantum mechanics the particle's wavefunction doesn't vanish in these regions, but rather monotonically decays. (For a constant potential, the decay is exponential.) This is called an evanescent wave, and has an interesting analogue in optics. But what if the total energy *exceeded* the potential energy? If the particle were initially moving to the right and encountered such an obstacle, according to classical mechanics it would slow down, but continue moving to the right.

EXERCISE 7.22

Consider the 200 eV step again, but with $E_0 = 225$ eV.

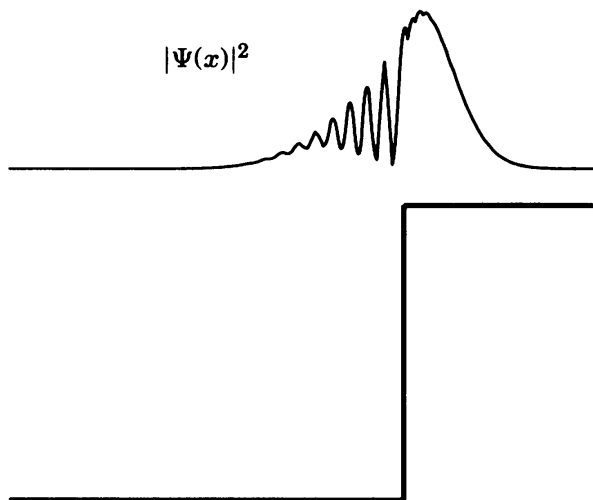


FIGURE 7.9 At 225 eV, the wavepacket's energy exceeds the 200 eV height of the step, yet we still observe interference-like behavior as a substantial portion of the wavepacket is being reflected.

Unlike classical mechanics, in quantum mechanics the wavepacket, or at least part of the wavepacket, will be *reflected* at the step while part of it is *transmitted*. If we wait until the transmitted and reflected components have well separated from one another, we can calculate the transmission and reflection coefficients as

$$\mathcal{T} = \frac{\int_0^{max} |\psi(x)|^2 dx}{\int_{min}^{max} |\psi(x)|^2 dx}, \quad (7.152)$$

$$\mathcal{R} = \frac{\int_{min}^0 |\psi(x)|^2 dx}{\int_{min}^{max} |\psi(x)|^2 dx}. \quad (7.153)$$

EXERCISE 7.23

Modify your code to calculate reflection and transmission coefficients, and investigate their relative magnitudes for $E_0 = 200, 225, 250, 275$, and 300 eV.

It's interesting to note that we can have reflection at a downward step as well. In this case, the kinetic energy of the wavepacket increases as it

encounters the (negative) step, but the wavepacket can still be reflected.

EXERCISE 7.24

Investigate the reflection and transmission coefficients for the case of a negative step, $V_0 = -200$ eV, for $E_0 = 25, 50, 75$, and 100 eV.

The Well

One of the classic examples of these processes, often reproduced in textbooks, is a wavepacket traveling over a well. That is, on the potential

$$V(x) = \begin{cases} 0, & x \leq 0, \\ V_0, & 0 \leq x \leq a, \\ 0, & a \leq x, \end{cases} \quad (7.154)$$

with V_0 negative, as shown in Figure 7.10. This example was originally discussed by Goldberg, Schey, and Schwartz, in “Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena,” *American Journal of Physics* **35**, 117 (1967). At the time, these were substantial calculations — but you can easily duplicate them, if you choose.

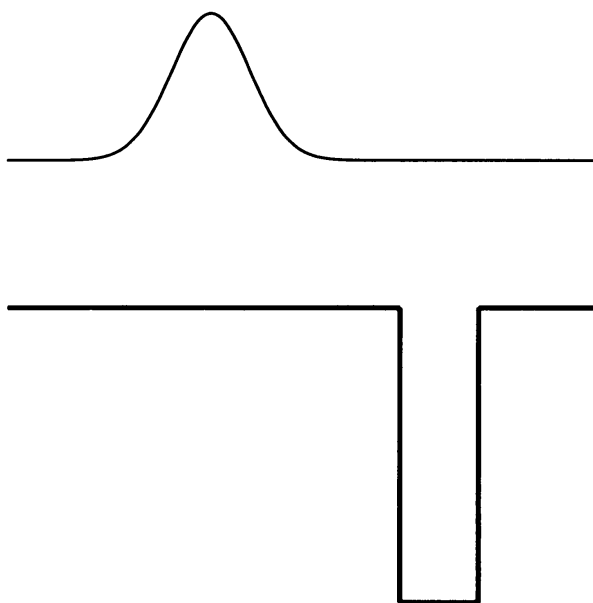


FIGURE 7.10 The wavepacket approaching a square well, with the parameters as discussed in the text.

Goldberg *et al.* used arbitrary units, but we can approximate their parameters if we take $E_0 = 100$ eV, $V_0 = -200$ eV, and $a = 1.963$ Å. With $m = 8$ and $-30 \text{ Å} \leq x \leq 30 \text{ Å}$, you will find that the k -space wavefunction is not well confined below the Nyquist limit, so that you will need to change some of the parameters in the calculation. The first change, of course, is to double the number of points — depending on how these points are distributed, the ranges of both x and k will change. Since the problem is in k -space, and not coordinate space, there is no reason to use a “balanced” approach, as we discussed earlier. Rather, you will want most of the extra range to appear in k -space. Let’s allow a modest increase in the x range, say to $-35 \text{ Å} \leq x \leq 35 \text{ Å}$, thereby almost doubling the k range. (Actually, it will be increased by about 71%.)

We need one more change before we’re ready to perform the calculations. For a free particle, i.e., the wavepacket away from the well, the energy is proportional to k^2 . We’ve just argued that the range of k had to be increased, almost doubled, in order to describe the wavepacket adequately. That is, the range in *energy* has been almost quadrupled! But an indicator of the error in the pseudo-spectral method is the quantity $[V, T]\delta_t$ — if T has been quadrupled and V unchanged, we need to reduce the time step by a factor of four!

Hint # 4: Choose a time step that is consistent with the highest energies that appear in the problem. In particular, as the range of k -space is increased, the range of energies is also increased, and the time step must be decreased correspondingly.

One more thing... Since the well will be sampled only a few times, e.g., the well is only a few h wide, the way the grid points at each end of the well are treated can become very important. Such *end effects* are often troublesome, and should always be considered with some care. The difficulty stems from the fact that the physical well and the computational grid need not be commensurate with one another, so that the ends of the well need not lie on the points of the grid. If we were then to increase the size of the grid, we would see no difference in the computation whatsoever — until, that is, the well extended to the next grid point, and then we would have a discontinuous change. To avoid these spurious effects, instead of just using $V(x_i)$, let’s use the potential averaged over h . Then, as the parameters of the calculation are changed we’ll realize a continuous change in the results, thus avoiding the spurious effects associated with the ends of the grid. Using a simple weighted averaging, we can devise a suitable code for the evaluation of $\text{Exp}V$:

```

left   = 0.000d0      ! Start of the well
right  = 1.963d0      ! End of the well
V0     = -200.d0      ! Magnitude of well, in eV
```

```

h2 = h / 2.d0

DO i = 1, n
  x = ...
  IF (x .le. left-h2) THEN
    V = 0.d0
  ELSEIF(x .gt. left-h2 .AND. x .le. left+h2) THEN
    V = V0 * (x-(left-h2)) / h
  ELSEIF(x .gt. left+h2 .AND. x .le. right-h2) THEN
    V = V0
  ELSEIF(x .gt. right-h2 .AND. x .le. right+h2) THEN
    V = V0 * (right+h2-x) / h
  ELSE
    V = 0.d0
  ENDIF
END DO

```

EXERCISE 7.25

Replicate the results of Goldberg *et al.* with the parameters changed as we've discussed. Follow the evolution for about 3×10^{-16} seconds.

“Simple” systems like this are very helpful in gaining a better understanding of the process. After you've convinced yourself that the program is running correctly, *and that you understand the results it's generating*, you might want to try different collision energies. Certainly, you might expect that the reflection and transmission coefficients will vary with E_0 , and that at sufficiently high energy the reflection will be small. But sometimes there are surprises along the way.

EXERCISE 7.26

Consider this scattering problem, with $E_0 = 150$ eV.

Can you explain the shape of the reflected wavepacket? The probability distribution in k space is also interesting, and might provide a clue. Essentially, what we have here is an exhibition of the *Ramsauer effect*, and we weren't even looking for it! Earlier, we found reflections from both a positive step and a negative one. What would you expect from the well? And how would you expect the two reflections to combine with one another?

The Barrier

Now, let's consider a barrier problem, that is, the potential given in Equation (7.154) with V_0 positive. You probably noticed in the exercises involving the positive step that the wavefunction penetrates into the “forbidden” region, e.g., the area where the potential is greater than the total available energy. If the barrier is sufficiently narrow, some of the wavefunction might “leak through” to the other side; that is, it might *tunnel* through the barrier. Goldberg *et al.* investigated this problem as well.

EXERCISE 7.27

With $E_0 = 100$ eV, $V_0 = +200$ eV, and $a = 1.93$ Å, replicate Goldberg's results for tunneling through a barrier.

Actually, with these parameters, there's not too much tunneling taking place — the barrier is just too high and too wide to allow the wavepacket through. A higher energy should permit more tunneling, of course.

EXERCISE 7.28

Try the tunneling problem with $E_0 = 200$ eV. The persistence of the wavepacket within the barrier is an interesting example of resonance in this physical situation.

And There's More...

This discussion of wavepackets is just an introduction to all that can be done with them, and all that can be learned from them in the pursuit of understanding in quantum mechanics. The ability to calculate these wavepackets opens up avenues that simply were not available when the quantum theory was being developed. While the fundamental physics is all contained in the Schrödinger equation, quantum mechanics is so rich that much is hidden within it, only to be teased out as we ask the right questions. For example, what if...

References

The solution of partial differential equations, particularly with finite difference methods, is a standard topic in many texts, including those we've previ-

ously referenced. The spectral method, however, is relatively new, e.g., not a product of nineteenth century mathematics. A definitive reference is

D. Gottlieb and S.A. Orszag, *Numerical Analysis of Spectral Methods: Theory and Applications*, SIAM-CBMS, Philadelphia, 1977.

Spectral methods are also discussed in some of the newer texts, such as

Gilbert Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, Massachusetts, 1986.

For an example of an important application, you might peruse

C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang, *Spectral Methods in Fluid Dynamics*, Springer-Verlag, New York, 1987.

C. A. J. Fletcher, *Computational Techniques for Fluid Dynamics*, Springer-Verlag, New York, 1988.

Although often used, the Baker-Campbell-Hausdorff theorem rarely receives the recognition it deserves.

J.E. Campbell, Proc. Lond. Math. Soc. **29**, 14 (1898); H.F. Baker, Proc. Lond. Math. Soc., Second Ser. **3**, 24 (1904); F. Hausdorff, Ber. Verh. Saechs. Akad. Wiss. Leipzig, Math.-Naturwiss. Kl. **58**, 19 (1906).

Appendix A:

Software Installation

Every computer has its own unique combination of hardware and software, and is managed by people with varying degrees of expertise. You might be using this textbook in a computer laboratory at a large university, in the library at a small college, or with your own computer in your dormitory. Given the wide range of possibilities, all we will attempt to do in this Appendix is to describe a basic configuration which works. *This information is primarily intended for the novice user — those with no or very little previous experience.* If you are using your own computer, make a complete backup of the system before you begin. It is extremely unlikely that any of the steps we will discuss will have any harmful effects, but it's far better to plan ahead for possible problems than to try to recover from unplanned disasters. If you are using someone else's microcomputer — the university's, for example, or your roommate's — DO NOT make any of the changes which we discuss without the permission of the person having responsibility for the computer. In any event, read this appendix thoroughly before making any changes.

In what follows, we assume that you are using a PC-compatible microcomputer in which an appropriate operating system and a Microsoft FORTRAN compiler have been successfully installed. If these have not yet been installed, then you should do so before proceeding.

Installing the Software

There are two distinct pieces of software that are distributed with *A First Course in Computational Physics*: the library and the code fragments. The library, creatively named `FCCP.lib`, contains various routines used in the text. Primarily, it serves as an interface between the text and the Microsoft graphics library. It also contains other routines to which you should have access, such as `ZIP_87` (discussed in Chapter 1) and `CT_PROJECTION` (discussed in Chapter 6). A User's Guide to the graphics routines is described in Appendix B, and a Technical Reference, including code listings, is provided in Appendix C.

We'll assume that your computer is equipped with a hard drive; let's call it C:. You will want to copy the library from the distribution diskette to the hard drive. If you followed Microsoft's standard installation guidelines, then you have a subdirectory named C:\FORTRAN\LIB. (Microsoft's graphics library, GRAPHICS.LIB, will have been stored here.) Although you may put the library wherever you choose, this seems like a reasonable place. For the moment, we'll assume that your computer is also equipped with an 80x87 numeric coprocessor. With the distribution diskette in drive A:, you can copy the library to the hard drive with

```
copy A:FCCP.LIB C:\FORTRAN\LIB
```

There is also a version of the library for computers without a numeric coprocessor. In that case you should use the slightly different copy command

```
copy A:FCCPno87.LIB C:\FORTRAN\LIB\FCCP.LIB
```

This copies the appropriate library to the hard drive and gives it the same name as before.

The distribution diskette also has several files containing "code fragments." Essentially, these are the lines of code as presented in the text. Rarely are these fragments complete programs — usually, they are only partially completed, but will hopefully start you in a constructive direction. You will probably want to copy these code fragments to the hard disk as well. We'll assume that you want to create a new subdirectory on drive C: for your work with *A First Course in Computational Physics*. This can be done with

```
makedir FCCP
```

The code fragments may then be copied from the distribution diskette to this new subdirectory with

```
copy A:\PIECES\*.* C:\FCCP
```

The files have names like 1.2, for example, which contains a code fragment pertinent to Exercise 1.2 of the text. As you begin this exercise, you will want give the file a new name. For example,

```
copy 1.2 try2.for
```

makes a copy of the code fragment and names it try2.for. Remember, all FORTRAN files must have an extension of FOR.

The FL Command

The Microsoft FORTRAN compiler and linker are invoked with the FL command. In its most basic form, the command is simply

```
fl try2.for
```

This command would compile `try2.for` and link it with the default library, producing an executable file `try2.exe`. However, we want to specify additional options to the compiler.

There are a large number of options that are available, all fully discussed in the FORTRAN Reference manual. We'll concentrate on just a few that have particular relevance to us. For example, in Chapter 1 we argued for variable names longer than 8 characters. However, the default option is to truncate variable names to just 8 characters. To override this default, we must specify the "No Truncation" option. The appropriate command is

```
fl /4Nt try2.for
```

The result is exactly the same as before, except that long variable names will not be truncated.

We also argued the case for strong typing. While the FORTRAN compiler will not *enforce* strong typing, the compiler will issue a warning message when it encounters an undeclared variable if it's invoked with the command

```
fl /4Yd try2.for
```

As novices, we should expect to make numerous errors, and be grateful for whatever help we can get. The compiler will issue extended error messages if we use the option

```
fl /4Yb try2.for
```

The messages are by no means verbose, but they are more helpful than the terse defaults.

These three compile options are the most important to us, but there are several others that might also be of interest. For example, if your computer has an Intel 80286 (or higher) processor, you can instruct the compiler to generate appropriate code with the option

```
fl /G2 try2.for
```

The default is to generate 8088 code, which is compatible with any of the Intel processors. That is, a program compiled *without* this option will run on any processor. A program compiled *with* the option will only run on 80286 or higher processors. However, a program compiled with the option and running on an 80286 processor should outperform the same program compiled without the option and running on the same machine. If you have an 80286 (or higher) processor, you should use this option.

To minimize total storage, the compiler normally puts data and instructions in the same storage area (segment) in computer memory. But as we attempt to solve more difficult and involved problems, we often find that we need rather large arrays to store all the necessary data. With the default options, several of the exercises in the text would exceed the capacity of the computer. To avoid this problem, we simply instruct the compiler to put the data into another segment with the option

```
f1 /Gt try2.for
```

We also need to instruct the compiler to use the appropriate libraries when linking. Assuming that FCCP.lib was copied to C:\FORTRAN\LIB, and that the Microsoft library GRAPHICS.lib was already in that subdirectory, the appropriate option is

```
f1 try2.for /link C:\FORTRAN\LIB\FCCP+
C:\FORTRAN\LIB\GRAPHICS
```

(Note: Due to the width of the page, two lines are needed to display this command line. You would use only one line if you were at the computer.) To instruct the compiler to use these libraries, as well as to not truncate variable names, issue extended error messages, issue warnings about undeclared variables, and use additional data segments, we need to invoke the compiler with the command

```
f1 /Gt /4Nt /4Yb /4Yd try2.for /link C:\FORTRAN\LIB\FCCP+
C:\FORTRAN\LIB\GRAPHICS
```

(If you have an 80286, 80386, or 80486 machine, you would also want the /G2 option.) This is a rather lengthy command, highly prone to numerous typing errors. Fortunately, there is an alternate way for us to provide these options to the compiler.

The FL environment variable can be used to specify frequently used options and commands. Fortunately, we don't need to know much about how this works. All we need to know is that the variable can be set with the com-

mand

```
set fl= /Gt /4Nt /4Yb /4Yd /link C:\fortran\lib\FCCP+
C:\fortran\lib\GRAPHICS
```

(Note that there is no space before the equals sign, and that the command would be entered as a single line at the computer.) Then, whenever the compiler is invoked with the simple

```
fl try2.for
```

the compiler will be invoked with all the options included in the environment variable. The process can be even further simplified if the `set` command is added to the `AUTOEXEC.BAT` file.

AUTOEXEC.BAT

This is a file that the computer uses when it's first turned on, and whenever it's rebooted. The commands in the file are executed just as in an ordinary BAT file, automatically. Some of those commands might be to execute programs, others might be to establish parameters in the environment. A typical file might look like the following:

```
path=.;c:\DOS;c:\FORTRAN\BIN
set lib=c:\FORTRAN\LIB
set tmp=c:\FORTRAN\TMP
set init=c:\DOS
```

These lines establish parameters in the environment: `path` tells the computer where to look for things; `lib` and `tmp` are used by the FORTRAN compiler; and `init` is used by various programs, such as your editor. We can edit this file and add the line

```
set fl= /Gt /4Nt /4Yb /4Yd /link C:\fortran\lib\FCCP+
C:\fortran\lib\GRAPHICS
```

These options will then be used whenever we compile a program. In our example the `lib` environment variable informs the compiler where to find libraries, so that the line

```
set fl= /4Nt /4Ybd /4Yd /link FCCP+GRAPHICS
```

accomplishes the same goal.

We hasten to add that another line might need to be appended as well. If your computer is equipped with a Hercules adapter, then you must run the program `MSHERC.COM` before using any graphics commands. This is easily done by adding the command

```
msherc
```

at the end of the `AUTOEXEC.BAT` file.

README.DOC

A summary of this Appendix is located in the file `README.DOC` on the distribution diskette. This file also contains a discussion of any revisions made to the software since the text was sent to the printers.

Appendix B:

Using FCCP.lib

Our interest in graphics is as a tool to enhance and promote our understanding of physics. For the most part, the graphics we'll use will be relatively simple and straightforward. The commands to draw lines and make simple drawings were briefly discussed in the text, primarily in Chapter 1, and are listed alphabetically and discussed more fully here. For more complicated figures or for a professional presentation, you might want to use a commercial software package specifically designed for graphics. Our intention is simply to provide an easy way of generating simple figures from within a FORTRAN program.

In this textbook, we rely upon Microsoft FORTRAN, Version 5.0 or later, which explicitly contains a graphics library, `GRAPHICS.LIB`, which we utilize. However, that library is much more extensive than we actually need, and so what we end up using is a rather small subset of all that is possible. In many ways the library is primitive, and uses conventions not typical of the physics community. For example, programmers in the graphics community insist on labeling the upper left-hand point on the display as the origin, and move *down* the screen as the y -coordinate increases. There are historical reasons for this, but any physicist *knows* that positive y should be directed *upward*.

There's also a question of the physical construction of the display device — how many pixels are there? (By the way, a *pixel* is a *picture element*, but why should we have to worry about these things!) To use the Microsoft graphics library effectively, you will need to know some of this — our graphics commands were written so that you would not need to interact quite so directly with the graphics library supplied by Microsoft. However, our commands certainly do not *exclude* you from using the library directly. And in fact, Microsoft's explanation of their graphics library, contained in the Microsoft FORTRAN Advanced Topics manual, is quite readable. (In comparison to similar manuals, at least!) The purpose of this Appendix is to describe our specific graphics commands — Appendix C shows how those commands were

implemented using the Microsoft graphics library.

Library User's Guide

The graphics commands are implemented in subroutines and functions, and are contained within `FCCP.lib`, the library distributed with *A First Course in Computational Physics*. The following User's Guide to those commands explains the commands and provides examples of their usage. Appendix C contains the explicit code that uses the Microsoft graphics library to implement these commands.

CLEAR This command clears the screen, in either text or graphics mode.

```
call CLEAR
```

If desired, after clearing the screen the cursor can be positioned by calling `CURSOR`. For example, the sequence

```
call CLEAR
call CURSOR( 1, 1 )
```

has the effect of moving the cursor to the upper left corner of the screen, the "home" position.

COLOR This subroutine changes the index of the currently active color.

```
call COLOR( INDEX )
```

`INDEX` is an integer variable identifying the active color. On black-and-white systems, the only possibilities are 0 (black) and 1 (white). On monochrome systems, the index refers to various possible grey scales. The value of `INDEX` ranges from zero to one less than the maximum number of available colors (or greys), as obtained from **NOC**.

CONTOURS This subroutine constructs contour lines from data supplied in an input array.

```
call CONTOURS( Nx, Ny, xx, yy, data,
+              Number_Contours, Values )
```

The array `DATA` holds the data from which the contour informa-

tion is to be obtained, dimensioned N_x by N_y , the number of grid points in the x - and y -directions. The arrays xx and yy contain the x - and y -coordinates of the grid lines. $Values$ is an array holding the values of the data for which contour lines are to be drawn, while $Number_Contours$ is the number of lines to be drawn.

CURSOR This subroutine moves the cursor to the specified row and column.

```
call CURSOR( row, column )
```

The variables `row` and `column` are declared as integer variables.

FILL This subroutine fills a designated area with the currently active color. This can be useful in providing a background color to a line plot.

```
call FILL( x1, y1, x2, y2 )
```

The coordinates specify the lower left corner (x_1, y_1) and upper right corner (x_2, y_2) of a rectangular region.

gEND This subroutine releases the graphics package and returns the computer to its standard configuration. This should be the last graphics command issued, as it erases the graphics image and returns the computer to text mode.

```
call gEND
```

gINIT This subroutine initializes the graphics package and the display device for producing graphics. It must be called before any other graphics command is executed.

```
call gINIT
```

After initialization, the viewport is the entire area of the display device. The subroutine `WINDOW` should be called establish the limits of the data to be mapped onto the viewport.

NOTE: If your computer is equipped with a Hercules adapter, you must run the `MSHERC.COM` program before attempting to display any graphics. See the Microsoft manual for further details.

LINE This subroutine draws a line from (x_1, y_1) to (x_2, y_2) .

```
call LINE( X1, Y1, X2, Y2 )
```

The variables must be declared Double Precision.

MAXVIEW This subroutine returns the size of the physical display, in pixels.

```
call MAXVIEW( NX, NY )
```

NX and NY are the number of pixels in the x - and y -dimensions, respectively, and are integer variables.

NOC This subroutine returns the number of possible colors (or grey scales) available.

```
call NOC( NUMBER )
```

NUMBER is an integer variable.

PIXEL This subroutine “turns on” a specific pixel with the currently active color. (The default color is white.)

```
call PIXEL( I, J )
```

I and J are integer variables identifying the I-th horizontal and J-th vertical pixel in the display.

VIEWPORT This subroutine specifies the region on the display device to be used. Scaled coordinates are used, so that the lower left of the screen is (0,0) and the upper right is (1,1).

```
call VIEWPORT( sx1, sy1, sx2, sy2 )
```

All variables are double precision.

The arguments of viewport were defined so as to make the placement of a viewport on the computer screen very convenient. In so doing, however, an additional complexity has been introduced that occasionally must be considered. The problem arises, for example, if you draw a circle. If you use

```
call VIEWPORT( .25d0, .25d0, .75d0, .75d0)
call WINDOW(-2.d0, -2.d0, 2.d0, 2.d0 )
```

and then call `line` to draw a circle, you will find that the figure is “squashed” since a unit of distance in the horizontal coordinate is not equivalent to one in the vertical coordinate.

WINDOW This subroutine maps data in the range $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$ onto the viewport of the display device.

```
call WINDOW( X1, Y1, X2, Y2 )
```

The variables must be declared Double Precision.

Appendix C:

Library Internals

We presume that most users will not need to know too much about what *actually* happens within graphics routines. In fact, the FCCP library was created so that you would not need to know how they work internally. However, some of you might want to know, or find yourself in need of this information, so we are providing it here. The ultimate source for the Microsoft graphics library is, of course, Microsoft's reference manuals.

One or more of the routines in our library can be combined into a single file for compiling. To interface properly with the Microsoft library, an initial line must be added,

```
include 'fgraph.fi'
```

This would then be the first line of the combined file.

Library Technical Reference

The graphics commands are implemented in subroutines and functions, and are contained within `FCCP.lib`, the library distributed with this text. The following Technical Reference illustrates how these commands are implemented using the Microsoft graphics library. We also include a detailed description of our contouring subroutine. A description of the commands and examples of their use are presented in the User's Guide of Appendix B.

CLEAR

This subroutine clears the screen in either graphics or text mode. It is implemented with the Microsoft library routine `clearscreen` using the symbolic constant `$GCLEARSCREEN`.

```

        Subroutine clear
*
* This routine clears the screen.  As currently
* implemented, it also moves the cursor to the HOME
* position.
*
        include 'fgraph.fd'
*
* Call Microsoft library subroutine:
*
        call clearscreen ( $GCLEARSCREEN )
        write(*,*)
        call cursor(1,1)

        end

```

COLOR

This routine selects the active drawing color, indexed by number.

```

        Subroutine color(number)
*
* This subroutine sets the active drawing color.
*
        include 'fgraph.fd'
        INTEGER*2 dummy
        integer number
*
* Call Microsoft library function:
*
        dummy = setcolor(number)
        end

```

Note that dummy is an INTEGER*2 variable, while number is an INTEGER variable. Our convention is that all real variables are double precision, and all integer variables are integer. This subroutine performs the type conversion as well as accessing the appropriate Microsoft library function.

CONTOURS

In general, drawing contours can be a difficult task, yet this routine rarely gets into trouble and usually produces nice contours. Considering its simplicity,

CONTOURS is a marvelous little program that works surprisingly well.

The primary input is the array DATA, containing the data from which the contours are to be constructed. There are `nx` data points in the x -direction, and `ny` data points in the y -direction, with the grid locations being stored in the arrays `xx` and `yy`, respectively. Note that the spacing need not be uniform, so that graded grids are easily considered. CONTOURS also needs to know how many contours are to be drawn, and with what values. Using `line`, we can draw the contours from within the subroutine.

The algorithm considers one square of the computational grid at a time. The location of the corners, e.g., the grid points at which data is available, are stored in arrays `X` and `Y`, and the values of the data at these points are stored in `VALUE`. The location of the center of the square and a simple approximation to the value of the data at the center are also stored. This fifth, central point divides the square into four triangles, which are examined one at a time.

The vertices of the triangle under consideration are indexed as `v1`, `v2`, and `v3`. These indices refer to where the data are stored in the `X`, `Y`, and `VALUE` arrays. The data are examined, and the variable `small` assigned the index of the smallest data value, `large` assigned the index of the largest data value, and `medium` assigned the remaining index. This is a crucial step in the algorithm, and one that is not particularly obvious. We do not want to *redefine* `v1`, for example, but rather we want to *define* `small`. Furthermore, `small` is not the smallest value, but the index that points to the smallest value. This reordering is done in the subroutine REORDER.

We can now determine the contour lines passing through this one triangle. Only if the value of the contour to be drawn is between the smallest vertex value and the largest vertex value will this triangle contain this contour. If it does, then inverse linear interpolation is used to find where the contour intersects the edge of the triangle. One edge must always lie between the smallest and the largest valued vertices — the intersection with this edge is found first. Which of the other two edges intersect the contour is determined by comparing the value of the contour to the value of `medium`, and the actual intersection determined by `INV`. The contour line is then drawn, and the next contour value considered. After all the contour values are considered, the next triangle is considered. After all the triangles in this grid square have been examined, the next grid square is considered, and so on, until all of the computational grid has been examined.

```
Subroutine contours( nx, ny, xx, yy, data,
+                  number_contours, contour)
```

```

*
* This subroutine constructs contour plots from given input data.
* Based on an article in the June 1987 issue of "BYTE" magazine
* by Paul D. Bourke and a subsequent comment in a following issue
* by Al Dunbar. Originally coded by Christopher J. Sweeney.
*
*                               originally written:    2/ 2/88 cjs
*                               last modified:         1/ 1/93 pld
*
*
*          nx          number of grid points in x-direction
*          ny          number of grid points in y-direction
*          xx          array containing x-coordinates of grid lines
*          yy          array containing y-coordinates of grid lines
*          data        array containing data to be displayed with
*                      contour plots
* number_contours    number of contour lines to be displayed
*          contour    array containing contour values
*
*****
*
* Declarations
*
*          integer nx, ny, number_contours
*          double precision xx(nx), yy(ny), data(nx,ny), contour(1)
*
*          double precision x(5), y(5), value(5),
* +                  x1, x2, y1, y2, target
*          integer lines, jx, jy, v1, v2, v3, small, medium, large
*
* The following STATEMENT FUNCTION 'inv' does inverse linear
* interpolation between the points (xa,ya) and (xb,yb) to find
* WHERE the function equals yyy.
*
*          double precision inv,  xa, ya, xb, yb,  yyy
*          INV(xa, ya, xb, yb, yyy) = xa+(yyy-ya)*(xb-xa)/(yb-ya)
*
* Start by looping over the data array. This is the "big" loop.
*
*          DO jy = 1,ny-1
*              DO jx = 1,nx-1
*
* We now store the x, y, and data values of the four corners
* and an approximation to the center in some temporary arrays.

```

* The points are numbered as

*

* 4 3

* 5

* 1 2

*

```
x(1)      = xx(jx)
y(1)      = yy(jy)
value(1) = data(jx,jy)
```

```
x(2)      = xx(jx+1)
y(2)      = yy(jy)
value(2) = data(jx+1,jy)
```

```
x(3)      = xx(jx+1)
y(3)      = yy(jy+1)
value(3) = data(jx+1,jy+1)
```

```
x(4)      = xx(jx)
y(4)      = yy(jy+1)
value(4) = data(jx,jy+1)
```

```
x(5)      = 0.5d0 * (x(1)+x(2))
y(5)      = 0.5d0 * (y(2)+y(3))
value(5) = 0.25d0 * ( value(1) + value(2)
+                      + value(3) + value(4) )
```

*

* Now we consider each of the 4 triangles, starting with 1-2-5
* and moving counterclockwise.

*

```
v3 = 5
DO v1 = 1,4
  v2 = v1 + 1
  if(v1 .eq. 4)v2 = 1
```

*

* At this point, we're considering one specific triangle.
* Arrange the vertices of this triangle to be in increasing
* magnitude.

*

```
      call reorder( v1, v2, v3,
+                  small, medium, large, value)
```

*

* Now we're ready to (maybe) draw some contours!

*

```

do lines = 1,number_contours
    target = contour(lines)
*
* But first, check to see if this contour value lies within this
* particular triangle:
*
    IF(value(small) .lt. target .AND.
+       target .lt. value(large) ) THEN
*
* O.K., the value of the contour lies between the smallest and
* largest values on this triangle -- we'll draw a contour!
*
* Now -- think about this -- one side of the triangle MUST
* contain BOTH the smallest and the largest vertices. Use
* inverse linear interpolation, INV, to find where the
* contour intersects that side.
*
    x1 = inv(x(small), value(small),
+          x(large), value(large), target)
    y1 = inv(y(small), value(small),
+          y(large), value(large), target)
*
* And now find the other side...
*
    IF (target .gt. value(medium)) THEN
+       x2 = inv(x(medium),value(medium),
+             x(large), value(large), target)
+       y2 = inv(y(medium),value(medium),
+             y(large), value(large), target)
    ELSE
+       x2 = inv(x(small), value(small),
+             x(medium),value(medium),target)
+       y2 = inv(y(small), value(small),
+             y(medium),value(medium),target)
    ENDIF

    call line( x1, y1, x2, y2 )

ENDIF
END DO
END DO
END DO
end

```

```

*
*-----
*
      Subroutine reorder ( i, j, k, small, medium, large, value)
      integer i, j, k, small, medium, large, temp
      double precision value(5)
*
* Do a "bubble sort" on the data. Since there are only 3
* items to be sorted, we'll write explicit code rather than
* using DO-loops.
*
      large = i           ! We need to start somewhere. Simply
      medium = j          ! make any assignments to initialize
      small = k           ! 'small', 'medium', and 'large'.

      IF(value(small) .gt. value(medium)) THEN
         temp = medium
         medium = small
         small = temp
      ENDIF

      IF(value(medium) .gt. value(large)) THEN
         temp = large
         large = medium
         medium = temp
      ENDIF
*
* The largest value has now bubbled to the top. Check
* that the next largest value is correct.
*
      IF(value(small) .gt. value(medium)) THEN
         temp = medium
         medium = small
         small = temp
      ENDIF
      end

```

CURSOR

This routine moves the cursor to the specified row and column.

```

      Subroutine cursor( row, column )
*

```

```

* Moves cursor to (row, column)
*
    include 'fgraph.fd'
    integer row, column
    INTEGER*2 i, j
    record / rccoord / curpos

    i = row
    j = column
*
* Call Microsoft library subroutine:
*
    call settextposition( i, j, curpos )

    end

```

FILL

This routine fills the interior of the specified rectangle with the current color. Useful in setting the backgrounds of line plots.

```

    Subroutine fill( x1, y1, x2, y2 )
*
* Fills the interior of the rectangle having lower
* left corner (x1,y1) and upper right corner (x2,y2)
* with the currently active color.
*
    include 'fgraph.fd'
    double precision x1,y1,x2,y2
    INTEGER*2 dummy
*
* Call Microsoft library function:
*
    dummy = rectangle_w( $GFILLINTERIOR, x1, y1, x2, y2 )
    end

```

gEND

This subroutine releases the graphics package and returns the computer to its default configuration. This should be the last graphics command issued, and it should always *be* issued, or else the computer is left in an undesirable state. This routine expects to read a null line before proceeding, e.g., it waits for you

to hit ENTER before it erases the screen, resets to text mode, and returns to the calling routine.

```

Subroutine gEND
*
* This is the last graphics command to be called. It
* waits for the ENTER key to be pressed, then resets the
* screen to normal before returning to the calling
* routine.
*
    include 'fgraph.fd'
    INTEGER*2 dummy
*
* Wait for the ENTER key:
*
    read (*,*)
*
* Call Microsoft library function:
*
    dummy = setvideomode( $DEFAULTMODE )
    end

```

gINIT

This subroutine initializes the graphics package and the display device. It must be called before any other graphics command is issued.

```

Subroutine gINIT
*
* This subroutine initializes the graphics package, and
* must be called before any other graphics command can
* be executed.
*
    include 'fgraph.fd'
    INTEGER*2          dummy, maxx, maxy, NUMBERofCOLORS
    RECORD /videoconfig/ myscreen
    COMMON / FCCP /      maxx, maxy, NUMBERofCOLORS
*
* Find graphics mode.
*
    call getvideoconfig( myscreen )

    SELECT CASE( myscreen.adapter )

```

```

        CASE( $CGA )
            dummy = setvideomode( $HRESBW )
        CASE( $OCGA )
            dummy = setvideomode( $ORESCOLOR )
        CASE( $EGA, $OEGA )
            IF( myscreen.monitor .EQ. $MONO ) THEN
                dummy = setvideomode( $ERESNOCOLOR )
            ELSE
                dummy = setvideomode( $ERESCOLOR )
            END IF
        CASE( $VGA, $OVGA, $MCGA )
            dummy = setvideomode( $VRES16COLOR )
        CASE( $HGC )
            dummy = setvideomode ( $HERCMONO )
        CASE DEFAULT
            dummy = 0
    END SELECT

    IF( dummy .EQ. 0 ) STOP 'Error:  cannot set graphics mode'
*
* Determine the maximum number of pixels in x- and y-directions.
*
    CALL getvideoconfig( myscreen )

    maxx = myscreen.numxpixels - 1
    maxy = myscreen.numypixels - 1
*
* And the maximum number of colors supported in this mode...
*
    NUMBERofCOLORS = myscreen.numcolors

    end

```

Much of this particular code is taken directly from the manual *Microsoft FORTRAN, Advanced Topics*.

LINE

This subroutine draws a line from (x_1, y_1) to (x_2, y_2) .

```

        Subroutine line( x1, y1, x2, y2 )
*
* This routine draws a line from (x1,y1) to (x2,y2).

```

```

*
      include 'fgraph.fd'
      double precision x1,y1,x2,y2
      INTEGER*2 dummy
      record / wxycoord/ wxy
*
* Call Microsoft library functions:
*
      call moveto_w(x1,y1,wxy)
      dummy = lineto_w(x2,y2)
      end

```

MAXVIEW

This subroutine returns the number of pixels in the x - and y -directions.

```

      Subroutine maxview( nx, ny )
*
* This subroutine returns the maximum number of pixels
* in x and y directions. This information was obtained
* when gINIT was called.
*
      INTEGER*2 maxx,maxy,NUMBERofCOLORS
      common / FCCP / maxx, maxy , NUMBERofCOLORS
      integer nx,ny

      nx = maxx
      ny = maxy
      end

```

Since the routine does not access a Microsoft library function, there's no need to include the file `fgraph.fd`.

NOC

This routine returns the number of colors (or levels of grey on monochrome systems) supported by the current computer configuration.

```

      Subroutine noc( number )
*
* This subroutine returns the maximum number of colors.
* This information was obtained when gINIT was called.

```

```

*
    INTEGER*2 maxx,maxy, NUMBERofCOLORS
    common / FCCP / maxx, maxy , NUMBERofCOLORS
    integer number

    number = NUMBERofCOLORS
end

```

PIXEL

While a very primitive task, it is sometimes desirable to “turn on” an individual pixel on the display. This routine does just that.

```

    Subroutine pixel( x, y )
*
* This subroutine turns on the pixel at (x,y), with
* the currently active color.
*
    include 'fgraph.fd'
    INTEGER*2 dummy
    integer x,y
*
* Call Microsoft library function:
*
    dummy = setpixel( x, y )
end

```

VIEWPORT

This routine selects a limited portion of the display device on which to draw. The region to be selected is specified in Scaled Coordinates, in which the lower left corner of the device is (0,0) and the upper right corner is (1,1).

```

    Subroutine viewport( x1, y1, x2, y2 )
*
* Set the VIEWPORT using Scaled Coordinates. The
* VIEWPORT must lie on the display device.
*
    include 'fgraph.fd'
    RECORD / videoconfig / vc
    double precision x1, x2, y1, y2
    INTEGER*2 ix1, ix2, iy1, iy2

```

```

*
* Call Microsoft library function:
*
    call getvideoconfig( vc )
*
* Convert to pixels, and verify that it's ON SCREEN
*
    ix1 = x1 * vc.numxpixels
    If( ix1 .lt. 0 ) ix1 = 0
    if( ix1 .gt. vc.numxpixels) ix1 = vc.numxpixels

    ix2 = x2 * vc.numxpixels
    if( ix2 .lt. 0) ix2 = 0
    if( ix2 .gt. vc.numxpixels) ix2 = vc.numxpixels

    iy1 = (1.d0-y1) * vc.numypixels
    if( iy1 .lt. 0) iy1 = 0
    if( iy1 .gt. vc.numypixels) iy1 = vc.numypixels

    iy2 = (1.d0-y2) * vc.numypixels
    if( iy1 .lt. 0) iy2 = 0
    if( iy1 .gt. vc.numypixels) iy2 = vc.numypixels
*
* Call Microsoft library subroutine:
*
    call setviewport( ix1, iy1, ix2, iy2 )

    end

```

WINDOW

This subroutine maps the data range onto the viewport.

```

    Subroutine Window( x1, y1, x2, y2 )
*
* Maps data in the range  $x_1 < x < x_2$  and  $y_1 < y < y_2$ 
* onto the viewport.
*
    include 'fgraph.fd'
    INTEGER*2 dummy
    double precision x1,y1,x2,y2
    logical*2 invert

```

```
        invert = .true.  
*  
* Call Microsoft library function:  
*  
        dummy = setwindow(invert,x1,y1,x2,y2)  
        end
```

Index

A

Air resistance, 217, 218, 224, 242
Airy, Sir George, 89
Aliasing, 318
Anaxagoras of Clazomenae, 149
Anharmonic oscillator, 236
ANIMATE, 347
Animation:
 pendulation, 234–236
 vibrating string, 347–353
Astrophysics, 338
Attenuation of x-ray beam, 326
Autocorrelation, 301
AUTOEXEC.BAT, 400
AVERAGE, 10

B

Back projection, 329
Backup copy, 8, 20
Backward difference, 110
Backward substitution, 123
BAK file, 8
Baker–Campbell–Hausdorff theorem, 379
Bessel function, 89
 derivatives of, 93
Best fit, *see* least squares
BISECT, 48
Bisection, 42–51
 combined with Newton-Raphson, 60–66
Boole's rule, 151
Boundary conditions, 237
 Dirichlet, 354
 irregular boundary, 359–361
 Neumann, 361–364
 ring magnet, 366
 quantum mechanical square well, 76, 78
Boundary value problem, 207

Butterfly, 29

C

Case insensitive, 5
CAT scan, 325
Center-of-mass, 243
Central difference, 110, 112
Central limit theorem, 196
Change of variables, 157–160, 163, 165, 168
Chirped pulse, 308
Clarity, 13–18
CLEAR, 357, 403, 407
Clever coding, 16
Code fragment, 11
COLOR, 26, 403, 408
Compile, 6
Compiling within an editor, 8
Computer graphics, 21–39
Computerized tomography, 325–338
Constants of the motion, 212–215
CONTOURS, 359, 403, 408–413
Convergence, rate of, 60–61, 71–74
Convolution, 294–298
Correlation, 298–309
Creating a file, 4
Crout decomposition, 121
Cubic splines, 95–108
CURSOR, 25, 404, 413
Curve fitting, *see also* interpolation
 by least squares, 117–119, 131–134
Curves in the plane, table of, 28–29

D

Debugging, 19
Deconvolution, 298
Default data types, 9
DEMO, 22
Derivatives, approximation of, 108–112, 252–253

Difference equations, 245
 Diffraction:
 circular aperture, 89
 knife's edge, 157
 Diffusion equation, 341
 Dirac delta function, 291
 Dirichlet's theorem, 281
 Discrete Fourier transform, 309–312
 Discretisation error, 250–254
 DOUBLE PRECISION variables, 9, 12–13
 Duffing's oscillator, 324

E

Earth–Moon system, 242–245
 Efficiency, 13, 16
 Eigenvalues, 257–260, 271–278
 Eigenvectors, 262–265
 Electrical networks, 130
 Elliptic differential equation, 342
 Elliptic integrals, 163–164
 Energy spectrum, 293
 Error flag, 65
 Errors:
 absolute, 47
 data entry, 11, 178–179
 discretisation, 250–254
 relative, 47
 round-off, 9
 truncation, 13
 Euler methods, 208–215
 Euler–McClaurin integration, 155
 Executable (EXE) file, 6
 Execution speed, 13
 Extrapolation, 91

F

False position, method of, 67–71
 Fast Fourier transform (FFT),
 312–316, 329–338
 FFT, 314–316, 335, 382–383
 FILL, 27, 404, 414
 Finite difference equations:
 heat equation, 354–357
 ring magnet, 367–369
 vibrating string, 344–353
 Finite elements, 265–278
 FL, 6, 398
 Flag index, 370

Forward difference, 110, 112
 FORTRAN, 2
 executing a program, 6
 getting started, 3
 source (FOR) file, 4–5
 Fourier series, 280–284
 Fourier transform, 284–294
 discrete, 309
 fast, 312
 multidimensional, 293–294
 properties of, 286
 Fourier, Jean Baptiste Joseph, 281
 Fractal, 32

G

Galerkin method, 269
 Gauss–Laguerre integration, 181
 Gauss–Legendre integration, 177, 273
 Gauss–Siedel iteration, 247–249
 Gaussian elimination, 119–131, 246
 Gaussian integration, 173–183
 Gaussian wavepacket, 383
 GEND, 21, 404, 414
 Gibbs phenomenon, 283
 GINIT, 21, 404, 415
 Gram–Schmidt orthogonalization, 173

H

Heat equation, 341
 steady-state, 354–358, 361
 HELLO, 7
 Hermite interpolation, 93
 Hilbert matrix, 129
 Hyperbolic differential equation, 342

I

Importance sampling, 195
 Initial value problem, 207
 Integration, 149–199
 composite rules, 152, 180
 Euler–McClaurin, 155
 Gaussian quadrature, 173–183
 Monte Carlo, 191–199
 multidimensional, 183–188
 Romberg, 155–157
 Simpson's rule, 151
 trapezoid, 151
 Romberg, 155–157

Interpolation:

- cubic spline, 103–108
- Lagrange, 86–89
- linear, 67
- Hermite, 93–95
- quadratic, 72

Isotherms, 358

Iteration, 55, 246

K

Kirchhoff's law, 295

Kronecker delta, 88

L

Lagrange interpolation, 86

Laplace equation, 341

Leakage, 318

Least squares, 117–119, 131–134,
266–267

- nonlinear, 138–147

- orthogonal polynomials, 134–137

Legendre polynomials, 189

- roots of, 57, 67, 176

LINE, 22, 404, 416

Link, 6

LOGICAL variables, 248–249

Log-log plots, 111, 204–205, 322

Logarithmic derivative, 240

Lorentzian lineshape, 138

LU decomposition, 121, 145

LUsolve, 125

M

Magnetic field of dipole, 364–374

MANDELBROT, 34

Mandelbrot set, 33

Matching point, 241

MAXVIEW, 34, 405, 417

Minimum uncertainty wavepacket, 384

Monte Carlo integration, 191–199

Monte Carlo simulation, 200–206

MONTE_CARLO, 192

N

Newton's second law, 244

Newton-Raphson method, 54–60

- combined with bisection, 60–66

NOC, 26, 405, 417

Noise, suppression of, 302, 306

Nonlinear least squares, 138

Normal equations, 132

Numeric coprocessor, 38, 397

Nyquist frequency, 317

O

Object (OBJ) file, 6

Ordinary differential equations, 207

Orthogonal polynomials, 137,

- 171–173, 189–191

- and least squares fitting, 134

- roots of, 175

- table of, 138

P

Parabolic differential equation, 342

Parseval's identity, 293

Pendulum, finite amplitude:

- motion of, 233

- period of, 160–164

Phase space, 231–234

PIXEL, 36, 405, 418

Poisson's equation, 341

Potential, hydrogen molecule, 241

Power method, 261–262

Power spectrum, 293

Production rules, 30

Program clarity, 13–18

Projectile motion, 217–218, 224, 242

Q

Quadrature, 150

Quantum mechanics, 74

R

RADAR, 304–309

Random number generators, 192

Random numbers, 191, 303, 305

RC circuit, 295

README.DOC, 401

Reflection at a potential step, 390

Residual error, 266

Responsibility for your work, 7

Richardson extrapolation, 113–117

RKF integrator, 221–224

- modified with TARGET, 320–321

Romberg integration, 155–157

ROOTS, 55, 62

Round-off error, 9

Runge–Kutta methods, 215–218

Runge–Kutta–Fehlberg, 219–226

S

Scattering, quantum mechanical,
388–394

Schrödinger equation, 75, 77, 236,
341, 377

Secant method, 70

Second-order differential equations, 226

Secular equation, 258

Separatrix, 234

Shooting method, 256

Signal-to-noise ratio, 302

Simpson’s rule, 151

SINE, 23

Snowflake, 31

Source code, 4

Spectral methods, 374

Spectrum analysis, 319

Spline, 106

SplineInit, 104

Square well, quantum mechanical:

double, 84

finite, 77–78

infinite, 74–77

scattering from, 391–393

Standard deviation, 197

Stochastic, 200

STRING, 345–347

String, vibrating, 254, 271–278,
342–353

Strong typing, 10

Structured programming, 14

Stubs, 15

Successive over-relaxation (SOR), 250

Symmetric expressions, 111

T

Taylor series, 51, 60, 86–88, 109,
150, 252, 349

Tomography, 325–338

Top-down design, 14

Transmission at a potential step, 390

Trapezoid rule, 151

Trial energy, 238

Tridiagonal linear systems, 99–103

TriSolve, 102

Truncation error, 13

TWO, 12

U

Undersampling of data, 318

Universal law of gravitation, 244

V

Van der Pol oscillator, 230, 232,
319–324

Vibrating string, 254, 271–278, 342–353

VIEWPORT, 21, 405, 418–419

Von Koch, Helge, 30

W

Wave equation, 340

Wavepacket, time evolution, 378

Weights and abscissas:

Gauss–Legendre quadrature, 177

Gauss–Laguerre quadrature, 181

Wiener–Khinchine theorem, 304

WINDOW, 22, 406, 419–420

X

X-rays, 325

Z

ZIP_87, 38, 396